# AC's TECH /AMIGA

*For The Commodore*

# Get Creative!

- PCX Graphics Viewer
- Understanding the Console Device
- Programming the Amiga in Assembly Language, Part 3
- CAD Application and Design, Part 4

## PLUS!

- Review of HiSoft's HighSpeed Pascal
- Programming the Amiga's GUI in C, Part 5
- Developer's Tools: Writing an Effective Press Release

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

**Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:**

AC's TECH Submissions
PiM Publications, Inc.
One Currant Place
Fall River, MA 02722

# Startup-Sequence

## The Amiga Market

I am often asked by fellow Amiga developers what I believe is the current state of the Amiga market. I appreciate the question, but I am not certain which one of my opinions I should give. When asked I am reminded of the story of the five blind men who were told to describe an elephant by feel. Each man went to a portion of the elephant and made a judgement of what the animal was like by what he felt.

"It is like a snake," said the man who was holding the trunk of the elephant.

"No, it is as solid and as stable as a tree," said the second man braced against the elephant's knee.

"You are both wrong. The elephant is a large flying animal with great wings," said a man feeling the elephant's large flexible ears.

"I feel an animal as big and solid as a wall," said the man who was feeling the elephant's side.

"This beast is like a camel or a great horse," said the blind man who had been placed on the back of the elephant's neck.

While these descriptions were not correct, they were not completely wrong. The blind men suffered from a problem of perspective. They knew only what they were allowed to feel and experience. They judged what they perceived based on their limited perspective. Their only fault was that they did not bother to check the elephant further. Their final decisions were based on incomplete information.

Many of us view the Amiga market from the same manner in which the blind men described their elephant. If we do not see advertising in the U.S., we assume that there is no advertising and that Commodore is abandoning the Amiga. It hardly crosses our minds that there are other advertising mediums than those we see. When members of CBM tell us that they are advertising in vertical markets and professional related areas, we complain that they are not doing more.

However, this scenario is currently playing in Commodore markets throughout the world. While Amiga users in North America feel that the Amiga is not being seen as the potential home/gaming computer suited for the commercial market as well as the professional arena, U.K. developers are upset that the Amiga is not being utilized as a professional platform, but merely as a game machine.

This fact is so entrenched in the psyche of people in the U.K. marketplace that I was currently told by a major executive of an English company that the Amiga was used only for games and that if anyone wanted to do anything serious with a computer, they would get an MS-DOS machine.

I walked into an outlet of a major electronics company in Europe and found the Amiga prominently displayed in an area designated for computer gaming. Across the room, where the more professional systems were, there were no Amigas, just MS-DOS machines. However, before anyone raises the point that this is the same in the U.S. due to the low prices for MS-DOS machines, I must tell you that the MS-DOS machines were selling for twice as much if not more than they are priced in the U.S. In addition, there were very few advanced MS-DOS units available. I found few 386 models and none of the 486 machines that have been selling inexpensively in the U.S. for some time now.

There is a lot to the point that at least the Amiga was available in a mass market setting and that the machine has established a solid footing. However, if people disregard the computer's better features, what future will the Amiga have?

From Sydney to Toronto, everyone has a different idea as to what the Amiga can do and how it should be marketed. While this poses a problem for Commodore International, it has offered a mother lode of opportunities for the rest of us. While everyone is pulling in different directions and establishing different markets for the Amiga, they have created a vast array of product openings.

With each new market, there is a need for more products. What about a wordprocessing, database, or spreadsheet product for the European market? It would be brightly colored, its features would be small but quickly understood, and it might even have arcade qualities built into it. It would be priced to sell competitively with the better games on the market, and it would make money.

In the U.S., look to see who is buying Amiga systems and develop more games for them. While I am assured by our staff statisticians that a great deal of professional systems are also being used by people who love to play games, I still would want to see a game that would interest people on a new level of sophistication so that it would not be confused with the more rapid arcade-style games currently available.

How about an entertainment package that made an arcade or strategy game out of video editing or sound manipulation? Why not use the interests of your market to create a better program?

CDTV is a masterful piece of hardware and Commodore is sincerely doing a better job of selling it than Phillips is doing in pushing CD Interactive. Everyone agrees that the platform is not the problem, but there needs to be a killer piece of software to sell the machine. Isn't this the market we are in? Isn't this all we need to be told to create the breaks we want?

Opportunity rarely comes hopping into the room with a bell around its neck. Opportunity sometimes occurs with the close scrutiny of a market, process, activity, or situation that eventually yields a new and better idea of attaining a goal. Opportunity can often occur with a single blind leap of inspiration, which I believe is the subconscious mind doing all the hard work while we end up taking the credit. But, no matter how the opportunity develops, it must be recognized for what it is.

Be realistic, you may discover only a small opportunity. Be thorough, it may be an opportunity that has already been fulfilled. Be tenacious, if you find it is something that could be done and you could do it—tear into it and get it done.

If you have a practical application that you believe could set the world on fire, then develop it. Don't wait for Commodore to establish the market. Commodore sells a platform that is called the Amiga. It is a great machine with a lot of capability. But the successes in this marketplace have not happened because they have waited for Commodore or any computer platform manufacturer to provide the market. They have occurred when individuals and companies have recognized the market for what it can become and have developed applications and tools to work within this new area.

Even with the mass marketing of MS-DOS machines and the flashy words from Apple Computer, I still have not seen the level of opportunity that I see in today's Amiga market—not only for the U.S. and Canada, but for the entire world. We are no longer a closed market. Our opportunities lie all around the globe. It only requires that we understand the entire elephant and not just the part we have hold of today.

Sincerely,
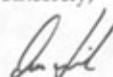
Don Hicks
Managing Editor

# PCX Graphics:
## Now You See Them!

*by Gary L. Fait*

Do you need an extra piece of clip art for your latest newsletter? How about some color artwork for your collection?

Take a look in the MS-DOS section of your favorite bulletin board system and you will probably find a large selection of graphics called PCX files. Of course, this may present a problem because PCX files are designed to be viewed (or created or changed) with *PC Paintbrush*, an MS-DOS program. Or is it really a problem?

You can view any 8-bit graphics file on your Amiga if you do a little research and a little programming. The accompanying program, ShowPCX, demonstrates how to read and view PCX files on the Amiga.

### What Are PCX Files?

PCX files are graphics files which have been stored in a format devised by ZSoft (Marietta, Georgia) when it created PC



This sample is actually consists of 16 colors.

Paintbrush, an all-purpose MS-DOS paint program. This file format has become a "standard" in the MS-DOS world, and most desktop publishing software provides PCX importing capabilities. In fact, many scanners output graphics in the PCX format, and both Apple and MS-DOS graphics programs often make use of that capability. That's good for us because it means there is a tremendous quantity of artwork out there waiting to be downloaded.

The PCX file format allows up to 256 colors to be used at once. To keep it simple, however, we will limit ShowPCX to 16 colors. But there are plenty of PCX files out there which use two colors (one bitplane) and 16 colors (four bitplanes), so you will have plenty to see.

### Reading a PCX File

In order to read a PCX file and display it on the Amiga, we need to follow this procedure:

1. Open the PCX file and read the header.
2. Open a SuperBitMap window.
3. Unpack a line of pixels from the file and store it in the SuperBitMap.
4. Repeat Step 3 until all lines have been read.
5. Open a window and display the file.
6. Allow user to scroll through the window and quit.

Almost any programming language can be used to read and display PCX files as long as you can access the Amiga's Intuition functions. This example uses the C language.

### The PCX Header

All PCX files begin with a 128-byte header. This header contains all the information necessary to read the file.

ShowPCX reads the header and stores the information in a structure we call pcx_hdr:

```
**************************************************************

struct   pcx_hdr

{

    unsigned   char   manufacturer;
    unsigned   char   version;
    unsigned   char   encoding;
    unsigned   char   bits_per_pixel;
    unsigned   int    xmin;
    unsigned   int    ymin;
    unsigned   int    xmax;
    unsigned   int    ymax;
    unsigned   int    hres;
    unsigned   int    vres;
    unsigned   char   palette[48];
    unsigned   char   reserved;
    unsigned   char   color_planes;
    unsigned   int    bytes_per_line;
    unsigned   int    palette_type;
    unsigned   char   filler[58];

};

**************************************************************
```

The structure members we will need to use are:

manufacturer—This unsigned char must equal 10, or it is not a PCX file.

xmin, ymin—These integers identify the upper left corner of the image (with 0,0 being the upper left corner of the window).

xmax, ymax—These identify the lower right corner.

palette[48]—Color information is stored in these 48 bytes. More about this later.

color_planes—This member will tell you the number of colors used to create the image. Since we get two colors for each bitplane, this number can be up to four, because we are limiting ShowPCX to 16 colors.

bytes_per_line—ShowPCX needs to know this so it knows how much "uncompressed" information to expect for each line of pixels in the image.

The other parts of the structure are either filler or are unnecessary for our purposes.

## The Program

Now that we understand the importance of the PCX header, let's look at the program. Much of the initial code in ShowPCX.c is used simply to lay the groundwork for the program. We identify the structures we are going to use, the types of variables which will be necessary, list the functions which are to follow, and create a simple "quit" menu.

As it is written, ShowPCX can be run from the CLI by typing:

                ShowPCX   graphicsfile

The program checks to see if your PCX file exists and then attempts to open it.

If everything is OK, the program then calls the function readheader(). The first step in this process is to read the first byte. If it does not equal 10, then it is not a PCX file and we exit with a brief message to the user. Assuming that the first byte equals 10, we continue to read the rest of the header variables.

By the way, you will notice a function in ShowPCX called readint(). Intel processors like those used in the IBM PC and clones store short integers "backwards." In an MS-DOS file, the least significant byte comes first, followed by the most significant byte. You can see that in readint() we read both bytes, shift the bits in the second byte to the left 8 bits and then "OR" them to get one 16-bit number we can use. It makes you appreciate owning an Amiga, doesn't it?

After reading the header, we need to check the color_planes member of pcx_hdr. If this number is more than 4, you have probably found a 256-color PCX file and ShowPCX won't read it. If the variable is 4 or less, we are in business!

We then decide if we need a lo-res screen or a hi-res screen and assign values to screenwidth, screenheight, windowwidth, and windowheight—all of which should need no explanation.

## Using a SuperBitMap Window

After opening the appropriate libraries, we can then set up for reading the actual PCX graphics data.

Most PCX files will not fit bit-for-bit on an Amiga screen. They are apt to be too high or too wide or both. Fortunately, the Amiga comes prepared for this. We can open a window which is actually larger than the screen and we can scroll the image to make it visible. This is called a SuperBitMap window, and the hard parts are taken care of by Intuition. (Note: For more about SuperBitMap windows, see "Scrolling Through SuperBitMap Windows" by Read Predmore in Amazing *Computing*, V4.1, January, 1989.)

We need to store the PCX data in our own bitmap and then open a window which uses that bitmap. The only limitation on the size and depth of the bitmap is the amount of memory in your Amiga.

ShowPCX creates a bitmap by calling the function openbm(). Since the PCX header told us the width, height, and number of color planes necessary, we feed that information to Intuition's InitBitMap() function. If you have enough memory, the Amiga handles all the dirty work. If there was a problem, we again notify the user and exit from the program.

Time for a brief review: So far, we have opened a PCX file, read the header, and created an empty bitmap. We have not opened a screen or window, nor have we read even the first byte of data from the file. So let's get going.

## Reading the File

For a one-bitplane image (two-color), the program calls the function domono(). The function docolor() is called for a four-bitplane image. Both work alike except that docolor() must handle four times as much data.

A two-color line of pixels is read with the function readline(). Since each bit of each byte is either "on" or "off," it is easy to read a line of pixels. Here's how:

PCX files use a crude form of compression called run-length encoding. The image is compressed by counting repeating bytes. If the top two bits of a byte are both 1, then the byte is a "count" byte. After disregarding the top two bits, the remaining value is

the number of times the following byte is repeated. The program code looks like this:

```
****************************************************************
do
{
    c = fgetc(sptr);                        /* Get a byte from the file */
    if((c & 0x3f);                          /* Are the top two bits set? */
    {
        i = (c & 0x3f);                     /* Get rid of the top two bits */
        c = fgetc(sptr);                    /* Get the next byte from file */
        while (i-)                          /* "Run" the byte */
        {
                pl[n] = c;
                ++n;
        }
    }
    else                          /* Else store the original byte */
    {
        pl[n] = c;
        ++n;
    }
} while(n < hdr.bytes_per_line);
        /* Continue until the line is full */
****************************************************************
```

A color line is very similar except that the four color planes are treated as one long line for each row of pixels. The readcolorline() function continues to read and "run" bytes until it has read four times the number of bytes we expect to find in a line.

For two- or 16-color images, the lines of pixels are stored in the SuperBitMap.

### Let's Take a Look!

Finally, we are ready to put something on the screen! A call to the function openwin() opens a screen and a window. It also creates our custom menu strip.

We also need to set the Amiga's colors to those used in the PCX file we are to view. We already read the color palette values from the PCX header and these are transferred to the screen with the function getpallette().

These values were stored in hdr.palette[]. We have 48 bytes of data—one byte for each red, green, and blue value for four bitplanes. We can go ahead and set our colors using all 48 bytes even if we are dealing with a two-color image.

At this point, there should be an image on the screen. But, as I mentioned earlier, a PCX image may be larger than the Amiga screen. That's why we used a SuperBitMap, remember?

To handle this, ShowPCX.c calls the functions getaction(), getmessage(), and getevent(). If the user moves the mouse pointer near the edge of the screen, the program will attempt to scroll the image in that direction. This is all explained in the source code and should be easy to follow.

To exit the program, the user chooses "Quit" from the menu. At that point, the program cleans up by closing everything it opened. As a final touch, ShowPCX prints out the values from the header to the screen.

### There's More

Because of space limitations, ShowPCX is pretty simple. But with a little work, any aspiring programmer can turn it into a masterpiece. One idea for improvement is adding the ability to handle 256-color PCX files. More "user friendliness" would also be nice, along with the ability to run the program from Workbench. Another idea is a routine to convert the images to IFF.

Well, what are you waiting for? Just be sure to share your results with the rest of us.

For more information on the PCX file format, see the following:

"Translating PCX Files" by Kent Quirk, Dr. Dobb's Journal, August, 1989.

Bit-Mapped Graphics by Steve Rimmer, Windcrest (Tab Books), 1990.

```c
/***************************************************************

ShowPCX.c

By Gary L. Fait
        Allows Amiga to display 2-color (one plane) and 16-
color (four planes) .pcx graphics files created with PC
Paintbrush (ZSoft, Marietta, GA).

****************************************************************/

#include  <exec/types.h>
#include  <intuition/intuition.h>
#include  <graphics/gfxmacros.h>
#include  <stdio.h>
#include  <stdlib.h>

/***************************************************************
        pcx.h
        Contains definition of a pcx header.
****************************************************************/
struct  pcx_hdr
{
        unsigned char  manufacturer;
        unsigned char  version;
        unsigned char  encoding;
        unsigned char  bits_per_pixel;
        unsigned int   xmin;
        unsigned int   ymin;
        unsigned int   xmax;
        unsigned int   ymax;
        unsigned int   hres;
        unsigned int   vres;
        unsigned char  palette[48];
        unsigned char  reserved;
        unsigned char  color_planes;
        unsigned int   bytes_per_line;
        unsigned int   palette_type;
        unsigned char  filler[58];
};

        /*** Other structures we will need ***/
struct  IntuitionBase *IntuitionBase;
struct  GfxBase *GfxBase;
struct  LayersBase *LayersBase;
struct  IntuiMessage *mesg,*GetMsg();
struct  NewScreen NewScreen;
struct  Screen *mainscreen;
struct  NewWindow NewWindow;
struct  Window *mainwindow;
struct  BitMap *mybitmap;
struct  pcx_hdr hdr;
struct  RastPort *rport;
struct  ViewPort *vport;
FILE *sptr, *fopen();

unsigned char source[128], k, c, *t, p1[512];
unsigned int gtemp, xtemp, finaltemp, quit, a, i, n, u, v,
        w, x, y;
unsigned long class, code;
short xscroll, yscroll, xmaxscroll, ymaxscroll, xpos, ypos,
        screenwidth, screenheight, screendepth,
windowwidth,
        windowheight;

/*** Functions we will use ***/
void  domono();
void  docolor();
void  readline();
void  readcolorline();
void  getpalette();
void  doscroll();
void  readheader();
void  printheader();
void  readint();
void  getaction();
void  getmessage();
void  getevent();
void  openlibs();
void  openbm();
void  openwin();
void  cleanup();

/***************************************************************
                        Set up menu
****************************************************************/
struct  IntuiText quittext =
{
        0,1,JAM1,CHECKWIDTH,0,NULL,"Quit",NULL
};

struct  MenuItem quititem =
{
        NULL,0,0,75,10,ITEMTEXT|HIGHBOX|ITEMENABLED,0,
        (APTR)&quittext,NULL,NULL,NULL,NULL
};

struct  Menu quitmenu =
{
        NULL,0,0,75,1,MENUENABLED,"Project",&quititem
};

/***************************************************************
        main()
****************************************************************/
main(argc,argv)
        int argc;
        char *argv[];
{
        printf("\nShowPCX..........\n");
        printf("By Gary L. Fait\n\n");
        printf("ShowPCX displays a PCX graphics
file.\n\n");

        /*** If necessary, show user how to run program.
***/

        if(argc < 2)
        {
        printf("\nTo run, use: ShowPCX
graphicfile.pcx\n\n");
        exit(0);
        }
        else    /*** Otherwise, user has done it right.
***/
        {
        strcpy(source,argv[1]);
        }

        /*** Add ".pcx" to filename if needed ***/
        if(strchr(source,'.') == NULL)
        {
        strcat(source,".pcx");
        }

        /*** Open file ***/
        if(!(sptr = fopen(source,"rb")))
        {
        printf("SORRY: Can't find/open %s.\n",source);
        exit(0);
        }

        readheader();   /*** Go read header information.
***/

        /*** Must not be more than 16 colors ***/
        if(hdr.color_planes > 4)
        {
        printf("\nThis file has more than 16 colors!\n");
        cleanup();
        }

        /*** Will it fit in a lores screen? ***/
        if(hdr.xmax <= 320)
        {
```

```c
    if(hdr.ymax <= 200)
    {
    screenwidth = hdr.xmax + 1;
    screenheight = hdr.ymax + 1;
    /*** Screen must be at least 320 pixels wide ***/
    if(hdr.xmax < 320)
    screenwidth = 320;
    screendepth = hdr.color_planes;

    windowwidth = hdr.xmax + 1;
    windowheight = hdr.ymax + 1;

    xmaxscroll = -1; /*** Prohibit scrolling ***/
    ymaxscroll = -1;
    }
    }
    /*** Should it be hires? ***/
    if(hdr.xmax > 320)
    {
    screenwidth = hdr.xmax + 1;
    screenheight = hdr.ymax + 1;
    /*** Screen must be at least 640 pixels wide ***/
    if(hdr.xmax < 640)
    screenwidth = 640;
    screendepth = hdr.color_planes;

    windowwidth = hdr.xmax + 1;
    windowheight = hdr.ymax + 1;
    /*** Don't go more than 1024 pixels high ***/
    if(windowheight > 1024)
    windowheight = 1024;

    /*** Determine max amount of scrolling allowed ***/
    xmaxscroll = windowwidth - 640;
    if(xmaxscroll < 0)
    xmaxscroll = 0;
    ymaxscroll = windowheight - 400;
    if(ymaxscroll < 0)
    ymaxscroll = 0;
    }

    openlibs();
    openbm();
    if(hdr.color_planes == 1) /*** If it's monochrome
***/
    {
    domono();
    }
    else if(hdr.color_planes == 4) /*** If it's 16-
color ***/
    {
    docolor();
    }
    else
    {
    cleanup();
    }
    openwin();
    getpalette();
    getaction();
    printheader();
    cleanup();
}


/**********************************************************
        domono()
        Reads monochrome pcx file into bitmap memory.
**********************************************************/
void domono()
{
    x = 0;
    for(y = 0; y < windowheight; y++)
    {
    readline();        /*** Get a line of data from file.
***/
    if(n < hdr.bytes_per_line)        /*** If it's EOF
***/
```

```c
    return;

    /*** Otherwise, copy line into the bitmap. ***/
    for(a = 0; a < hdr.bytes_per_line; a++)
    {
    mybitmap->Planes[0][x] = p1[a];
    x++;
    }
    }
}


/**********************************************************
        docolor()
Reads 16-color pcx file into bitmap memory.
NOTE: Z Soft uses an interleaved format for color images.
**********************************************************/
void docolor()
{
    u = 0;
    v = 0;
    w = 0;
    x = 0;
    for(y = 0; y < windowheight; y++)
    {
    /*** Get line of data from file. ***/
    /*** Actually, you will get a 'scan line' of
data...
        which will include enough data to fill a
line
        in each bit plane. ***/
    readcolorline();

    if(n < 4 * hdr.bytes_per_line)        /*** If it's EOF
***/
    return;

    a = 0;
    while(a < (n - 1))
    {
    /*** Store a line of data in first bitplane ***/
    for(i = 0; i < hdr.bytes_per_line; i++)
    {
    mybitmap->Planes[0][u] = p1[a];
    ++u;
    ++a;
    }
    /*** Store a line of data in second bitplane ***/
    for(i = 0; i < hdr.bytes_per_line; i++)
    {
    mybitmap->Planes[1][v] = p1[a];
    ++v;
    ++a;
    }
    /*** Store a line of data in third bitplane ***/
    for(i = 0; i < hdr.bytes_per_line; i++)
    {
    mybitmap->Planes[2][w] = p1[a];
    ++w;
    ++a;
    }
    /*** Store a line of data in third bitplane ***/
    for(i = 0; i < hdr.bytes_per_line; i++)
    {
    mybitmap->Planes[3][x] = p1[a];
    ++x;
    ++a;
    }
    }
    }
}


/**********************************************************
        readline()
        Read and decompress a single line from file.
        Use this function for monochrome files.
**********************************************************/
void readline()
```

```
{
        n = 0;
        i = 0;

        do
        {
        c = fgetc(sptr);            /*** Get a byte ***/

                                    /*** If it's a run of bytes field
***/
        if((c & 0xc0) == 0xc0)
        {
        i = (c & 0x3f);  /*** AND off the high bits ***/
        c = fgetc(sptr);            /*** Get the run byte
***/
        while(i--)
        {
        pl[n] = c;                          /*** Run the byte
***/
        ++n;
        }
        }
        else
        {
        pl[n] = c;       /*** Else store the original byte
***/
        ++n;
        }
        } while(n < hdr.bytes_per_line);
}


/*************************************************************
        readcolorline()
        Read and decompress 16-color line from file.
This function will read an entire 'scan line' of data. It
will contain the data to fill all four bitplanes for one
color line on the screen.
*************************************************************/
void  readcolorline()
{
        n = 0;
        i = 0;

        do
        {
        c = fgetc(sptr);            /*** Get a byte ***/

                                    /*** If it's a run of bytes field
***/
        if((c & 0xc0) == 0xc0)
        {
        i = (c & 0x3f);  /*** AND off the high bits ***/
        c = fgetc(sptr);            /*** Get the run byte
***/
        while(i--)
        {
        pl[n] = c;                          /*** Run the byte
***/
        ++n;
        }
        }
        else
        {
        pl[n] = c;       /*** Else store the original byte
***/
        ++n;
        }
        } while(n < (4 * hdr.bytes_per_line));
}


/*************************************************************
                    getpalette()
Reads 16 RGB colors from palette and sets screen colors.
It's okay to use this for monochrome files, too, but only
the first two colors will be used.
*************************************************************/
void  getpalette()
```

```
{
        char red, green, blue;
        a = 0;

        for(i = 0; i < 16; i++)
        {
        /*** Shift each value right four bits ***/
        red = hdr.palette[a] >> 4;
        a++;
        green = hdr.palette[a] >> 4;
        a++;
        blue = hdr.palette[a] >> 4;
        a++;
        SetRGB4(vport,i,red,green,blue);
        }
}


/*************************************************************
                    doscroll()
Takes information from getaction function and moves image on
screen ... if possible.
*************************************************************/
void  doscroll()
{
        /*** If we're already at left edge, don't move.
***/
        if((xscroll + xpos) < 0)
        return;

        /*** If we're already at top, don't move. ***/
        if((yscroll + ypos) < 0)
        return;

        if(xscroll != 0)
        {
        /*** If we're at right edge, don't move. ***/
        if((xpos + xscroll) > xmaxscroll)
        {
        xscroll = 0;
        }
        else
        {
        xpos = xpos + xscroll;
        }
        }
        if(yscroll != 0)
        {
        /*** If we're at bottom, don't move. ***/
        if((ypos + yscroll) > ymaxscroll)
        {
        yscroll = 0;
        }
        else
        {
        ypos = ypos + yscroll;
        }
        }
        /*** If we've made it this far, go ahead and scroll
the
        screen. ***/
        ScrollLayer(mainwindow->WLayer->LayerInfo,
        mainwindow->WLayer,xscroll,yscroll);
}


/*************************************************************
                    readheader()
        Read header information from beginning of file.
*************************************************************/
void  readheader()
{
        printf("\nReading PCX file ... Please wait.\n");
        fseek(sptr,0L,0);
                /*** Make sure this is a .pcx file. ***/
        fread((char *) &hdr.manufacturer,1,1,sptr);
        if(hdr.manufacturer != 10)
        {
        printf("ERROR: Not a PCX file.\n");
```

```c
    cleanup();
    }

    /*** Next byte is ZSoft PC Paintbrush version. ***/
    fread((char *) &hdr.version,1,1,sptr);
    /*** Now convert to string. ***/
    switch(hdr.version)
    {
    case 0:
    t = "2.5";
    break;
    case 2:
    t = "2.8 with pallette info";
    break;
    case 3:
    t = "2.8 without pallette info";
    break;
    case 5:
    t = "3.0";
    break;
    default:
    t = "UNKNOWN";
    break;
    }

    /*** Next byte is encoding method ***/
    fread((char *) &hdr.encoding,1,1,sptr);

    /*** Next byte is bits_per_pixel ***/
    fread((char *) &hdr.bits_per_pixel,1,1,sptr);

    /*** Next 2 bytes are xmin ***/
    readint();
    hdr.xmin = finaltemp;

    /*** Next 2 bytes are ymin ***/
    readint();
    hdr.ymin = finaltemp;

    /*** Next 2 bytes are xmax ***/
    readint();
    hdr.xmax = finaltemp;

    /*** Next 2 bytes are ymax ***/
    readint();
    hdr.ymax = finaltemp;

    /*** Next 2 bytes are hres ***/
    readint();
    hdr.hres = finaltemp;

    /*** Next 2 bytes are vres ***/
    readint();
    hdr.vres = finaltemp;

    /*** Next 48 bytes are palette ***/
    fread((char *) &hdr.palette,48,1,sptr);

    /*** Next byte is vmode (reserved) ***/
    fseek(sptr,1,1);

    /*** Next byte is color_planes ***/
    fread((char *)&hdr.color_planes,1,1,sptr);

    /*** Next 2 bytes are bytes_per_line ***/
    readint();
    hdr.bytes_per_line = finaltemp;

    /*** Next 2 bytes are palette_type ***/
    readint();
    hdr.palette_type = finaltemp;

    /*** Next 58 bytes is filler***/
    fread((char *) &hdr.filler,58,1,sptr);

}

/************************************************************
```

```c
                printheader()
        Prints header information to screen.
************************************************************/
void printheader()
{
    /*** Print header information to CLI screen ***/
    printf("\nInformation from %s:\n\n",source);
    printf("Created with PC Paintbrush version
%s.\n",t);
    printf("Compression method: ");
    if(hdr.encoding != 1)
    {
    printf("None.\n");
    }
    else
    {
    printf("Run Length Encoding.\n");
    }
    printf("Number of bits per pixel = %u.\n",
    hdr.bits_per_pixel);
    printf("Begins at (%u, %u) and ends at (%u,
%u).\n",
    hdr.xmin,hdr.ymin,hdr.xmax,hdr.ymax);
    printf
    ("Created on a device with %u x %u dpi
resolution.\n",
    hdr.hres,hdr.vres);
    printf("Image contains %u planes of data.\n",
    hdr.color_planes);
    printf("Uncompressed line has %u bytes of data.\n",
    hdr.bytes_per_line);
    printf("Image palette is ");
    if(hdr.palette_type = 1)
    {
    printf("color.\n");
    }
    else if(hdr.palette_type = 2)
    {
    printf("gray./n");
    }
    else
    {
    printf("unknown.\n");
    }
    printf("Uncompressed file size = %u.\n\n\n",
    ((unsigned long)hdr.ymax - (unsigned long)hdr.ymin
+ 1)
    * (unsigned long)hdr.bytes_per_line *
    (unsigned long)hdr.color_planes);
}

/************************************************************
            readint()
Intel processors store short integers "backwards." The least
significant byte comes first, followed by the most signifi-
cant byte. We have to read them and put them in the right
order.
************************************************************/
void readint()
{
    fread((char *) &k,1,1,sptr);        /*** Read first
byte ***/
    xtemp = (unsigned int) k;
    fread((char *) &k,1,1,sptr);        /*** Read next
byte ***/
    gtemp = (unsigned int) k;
    gtemp <<= 8;    /*** Shift 2nd byte left by 8 bits
***/
    finaltemp = (xtemp |= gtemp);       /*** OR them
***/
}

/************************************************************
            getaction(), getmessage(), getevent()
        These functions check for and act on user input.
************************************************************/
void getaction()
```

```c
{
        while(quit == 0)
        {
        getmessage();
        getevent();
        }
}

void getmessage()
{
        class = 0;
        code = 0;

        Wait(1 << mainwindow->UserPort->mp_SigBit);
        while(mesg = GetMsg(mainwindow->UserPort))
        if(mesg != NULL)
        {
        class = mesg->Class;
        code = mesg->Code;
        ReplyMsg(mesg);
        }
}

void getevent()
{
        if(class == MENUPICK)
        {
        if(code == MENUNULL)                /*** It's not a
pick ***/
        return;

        if(MENUNUM(code) == 0)    /*** It's the Project
menu ***/
        {
        if(ITEMNUM(code) == 0)               /*** Quit ***/
        {
        quit = 1;
        return;
        }
        }
        }
        /
/****************************************************************
        If the mouse pointer is near the edge of the
screen, assume that user wants to scroll in that direction.

****************************************************************/
        while(mainscreen->MouseY < 20)
        {
        xscroll = 0;
        yscroll = -5;
        doscroll();
        }
        while(mainscreen->MouseY > 380)
        {
        xscroll = 0;
        yscroll = 5;
        doscroll();
        }
        while(mainscreen->MouseX < 20)
        {
        yscroll = 0;
        xscroll = -5;
        doscroll();
        }
        while(mainscreen->MouseX > 620)
        {
        yscroll = 0;
        xscroll = 5;
        doscroll();
        }
}

/****************************************************************
        openlibs()
                Opens libraries required by ShowPCX.
****************************************************************/
```

```c
void openlibs()
{
        IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library",0);
        if(IntuitionBase == NULL)
        {
        cleanup();
        }

        GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library",0);
        if(GfxBase == NULL)
        {
        cleanup();
        }

        LayersBase = (struct LayersBase *)
        OpenLibrary("layers.library",0);
        if(LayersBase == NULL)
        {
        cleanup();
        }
}

/****************************************************************
                        openbm()
Creates bitmap to be used by SuperBitMap window. Allocates
memory for each plane according to information found in the
pcx header.
****************************************************************/
void openbm()
{
        mybitmap = malloc(sizeof (struct BitMap));
        if(mybitmap != 0)
        {
        InitBitMap(mybitmap,hdr.color_planes,windowwidth,
        windowheight);
        for(i = 0; i < hdr.color_planes; i++)
        {
        if((mybitmap->Planes[i] =
        (PLANEPTR) AllocRaster(windowwidth,windowheight))
        == NULL)
        {
                printf("Not enough memory for bitmap!");
                cleanup();
        }
        BltClear(mybitmap->Planes[i],
        RASSIZE(windowwidth,windowheight),0);
        }
        }
        else
        {
        printf("Can't allocate memory for bitmap!");
        cleanup();
        }
}

/****************************************************************
                        openwin()
Opens screen, window, RastPort, ViewPort and menu strip.
****************************************************************/
void openwin()
{
        NewScreen.LeftEdge = 0;
        NewScreen.TopEdge = 0;
        NewScreen.Width = screenwidth;
        NewScreen.Height = screenheight;
        NewScreen.Depth = screendepth;
        NewScreen.DetailPen = 0;
        NewScreen.BlockPen = 1;
        NewScreen.ViewModes = NULL;
        if(screenwidth <= 320)
        NewScreen.ViewModes = NULL;
        if(screenwidth > 320)
        NewScreen.ViewModes |= HIRES;
        if(screenheight > 200)
        NewScreen.ViewModes |= LACE;
```

```c
        NewScreen.Type = CUSTOMSCREEN;
        NewScreen.Font = NULL;
        NewScreen.DefaultTitle = NULL;
        NewScreen.Gadgets = NULL;
        NewScreen.CustomBitMap = NULL;

        if((mainscreen = (struct Screen
*)OpenScreen(&NewScreen))
        == NULL)
        {
        printf("\nNot enough memory to open screen!\n");
        cleanup();
        }
        ShowTitle(mainscreen,FALSE);

        NewWindow.LeftEdge = 0;
        NewWindow.TopEdge = 0;
        NewWindow.Width = windowwidth;
        NewWindow.Height = windowheight;
        NewWindow.DetailPen = 0;
        NewWindow.BlockPen = 1;
        NewWindow.Title = NULL;
        NewWindow.Flags = SUPER_BITMAP|BORDERLESS|ACTIVATE|
        REPORTMOUSE;
        NewWindow.IDCMPFlags =
MENUPICK|ACTIVEWINDOW|MOUSEMOVE;
        NewWindow.Type = CUSTOMSCREEN;
        NewWindow.FirstGadget = NULL;
        NewWindow.CheckMark = NULL;
        NewWindow.Screen = mainscreen;
        NewWindow.BitMap = mybitmap;
        NewWindow.MinWidth = 30;
        NewWindow.MinHeight = 30;
```

```c
        NewWindow.MaxWidth = windowwidth;
        NewWindow.MaxHeight = windowheight;

        if((mainwindow = (struct Window
*)OpenWindow(&NewWindow))
        == NULL)
        {
        printf("\nNot enough memory to open window!\n");
        cleanup();
        }
        rport = mainwindow->RPort;
        vport = &mainwindow->WScreen->ViewPort;

        SetMenuStrip(mainwindow,&quitmenu);
}

/******************************************************************
        cleanup()
        Closes anything and everything we have opened.
******************************************************************/
void cleanup()
{
        if(sptr)
        fclose(sptr);

        for(i = 0; i < hdr.color_planes; i++)
        {
        if(mybitmap->Planes[i])
        FreeRaster(mybitmap->Planes[i],windowwidth,
        windowheight);
        }

        if(mybitmap)
        {
        free(mybitmap);
        setmem(mybitmap,sizeof(mybitmap),0L);
        }

        ClearMenuStrip(mainwindow);

        if(mainwindow)
        CloseWindow(mainwindow);

        if(mainscreen)
        CloseScreen(mainscreen);

        if(LayersBase)
        CloseLibrary(LayersBase);

        if(GfxBase)
        CloseLibrary(GfxBase);

        if(IntuitionBase)
        CloseLibrary(IntuitionBase);

        exit(0);
}
```

☑

**Please Write to:**
**Gary L. Fait**
**c/o AC's TECH**
**P.O. Box 2140**
**Fall River, MA 02722-2140**

# HiSoft's HighSpeed Pascal

*by David Czaya*

*HighSpeed Pascal* (v1.00) from HiSoft, is an aptly named Pascal IDE (Integrated Development Environment) for the Amiga which closely adheres to the programming characteristics of *TurboPascal* by Borland Inc.

The HSPascal environment consists of a text editor which gives complete control over editing, compiling, debugging, and running your programs. For people who wish to use their own editor or run things from the CLI, the compiler and debugger are runnable as standalone programs.

Installation, though not automated, can be accomplished by moving a few directories and program files. This is all very well documented in the manual, and icons are provided for Workbench use.

## TurboPascal Compatibility

HighSpeed Pascal has a high degree of TurboPascal v5.0 compatibility. This includes support for dynamic length strings, DOS, CRT, and Graph units, standard function names and parameter lists, as well as other TurboPascal extensions to the Pascal language.

The compatibility does not include any of the *TurboVision* OOP (Object Oriented Programming) or Windows concepts available in TurboPascal v5.5 and above.

I have compiled many complex TurboPascal programs without a single change. The most predominant incompatibility occurs when the source code uses hardware specific programming (mouse, inline code, assembler, etc.). Also, it's noted that data files written for TurboPascal may need the byte order swapped before using with HSPascal because of the manner in which the 680x0 and the 80x86 machines store data.

Typically, when the programmer uses the TP units for text, graphics, and DOS, the source code will compile and run just fine. This, in itself, is immensely exciting considering the abundance of TurboPascal source code available online, on disk, and in reference books.

## Programming with HighSpeed Pascal

There are many books teaching and referencing TurboPascal and any one of them can be used with HighSpeed Pascal. HSPascal uses the "unit" concept which made TurboPascal stand out above the rest. A unit is similar to a module in Modula 2 or a library in C. It allows you to create functions and procedures which may be included in other programs rather than rewriting the code within each program.

HSPascal (as well as TurboPascal) include powerful functions to simplify certain programming tasks. The DOS unit supplies functions for file I/O, directory, and command line handling, and date-time manipulation.

The CRT unit provides console window procedures including keyboard I/O, text styles, cursor placement, etc.

Finally, the Graph unit provides TurboPascal compatible routines for screen resolution, color palette, font styles, text justification, graphic fill patterns, aspect ratios, 2-D and 3-D bar graph drawing, pixel, line, poly, and image drawing.

HSPascal is very similar to ISO and ANSI Pascal although there are some rather insignificant differences and extensions to the language.

HSPascal can access any Amiga library, device, or resource. An included utility converts standard Amiga function definition (.fd) files to HSPascal units. Each Amiga system function has been converted in this manner (v2.04 units will be available by the time you read this). In addition, all types, and constants from the Amiga C language "include" files have been added giving complete access to the Amiga system. The function names, calling conventions, and constants all use the C names and syntax wherever possible.

HSPascal includes an inline assembler supporting the full 68000 instruction set—with minor compiler restrictions. This allows you to write regular assembler statements using assembler operators and directives mixed with standard Pascal labels, constants, and variables. This is a very powerful concept for easily adding assembly language routines to your code.

A programming problem you will encounter has to do with Pascal text strings. Pascal uses a length byte followed by the string's characters (essentially the same as BCPL strings). HiSoft has added two functions, PasToC and CToPas, to cope with conversions to C style, null-terminated strings which you will need when working with the Amiga.

Another potential problem inherent to Pascal is that of allocating memory off the heap with New or GetMem. There are no provisions to specify memory type on the Amiga (i.e., Chip, Fast, etc.).

### The IDE Editor

HighSpeed Pascal includes separate editors for AmigaDOS 1.3 and 2.0 users. They appear and function identically so that you simply choose whichever applies when installing HSPascal.

The editor has an intuitive design that conforms closely to Release 2-style guidelines. It runs fast and has good mouse, keyboard, and menu control. The editor fully supports the clipboard copy-cut-and-paste, an extensive print facility, undo and undelete line, fast find and replace, bookmarks, and more. There is even a recorder-style macro player to perform repetitive editing tasks.

All environmental settings can be made from the editor with gadgets, path requesters, check, radio, and cycle gadgets. You can compile, set program arguments, debug, and run your programs from simple menu commands or logical keyboard equivalents. Nearly everything can be controlled from the keyboard or the mouse even within special requesters.
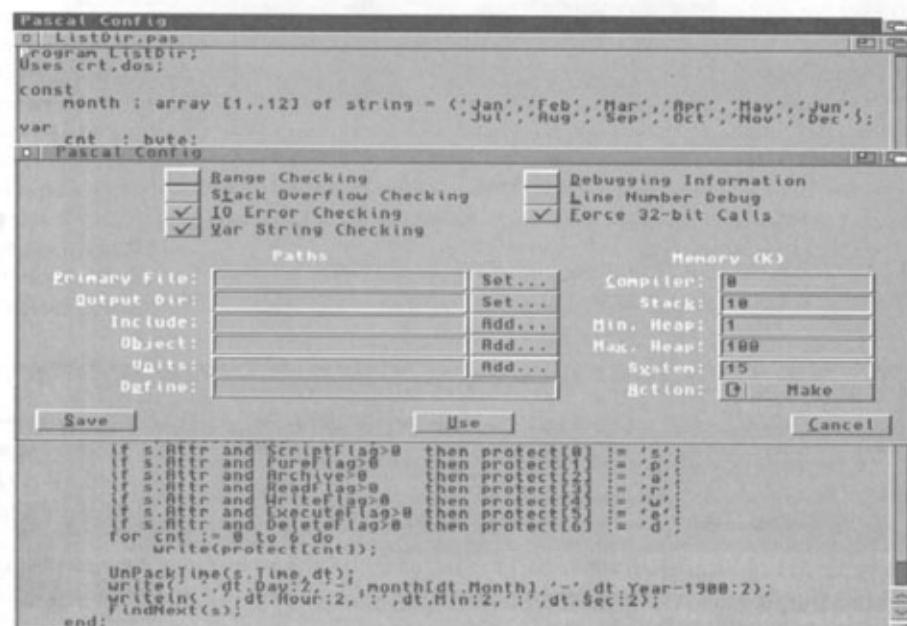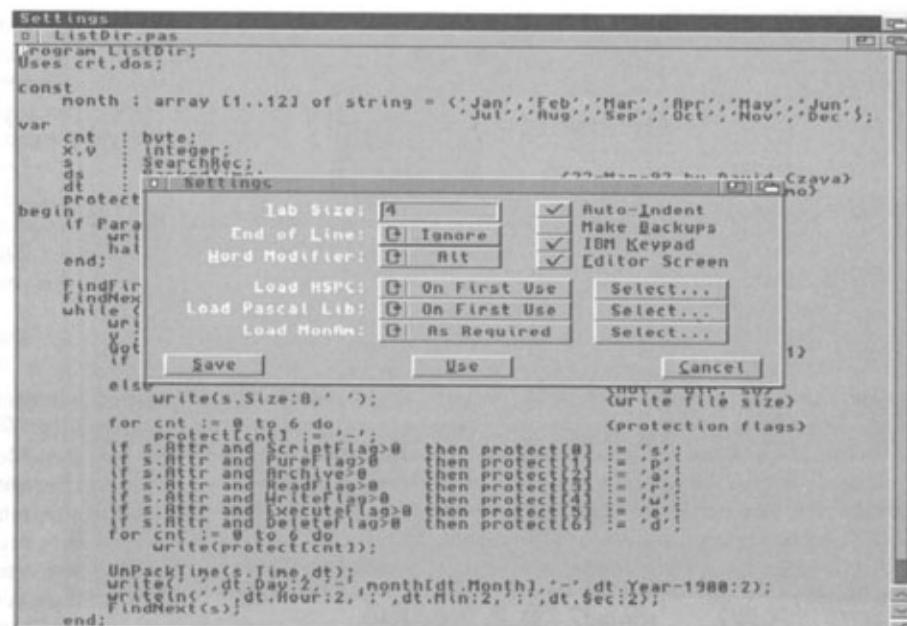
Strangely absent are some of the more basic editor functions such as case conversion and character swap. Also, there is no ARexx interface provided.

Compilation errors are flagged first in a popup requester, and then in the editor's title bar. The cursor is placed on the first error ready for correction.

### The Compiler

HSPC, the compiler, is a single-pass compiler/linker. It includes standard options for range and stack checking and inserting debug symbols as well as I/O and var-string error checking. The compiler manages your projects with a built-in "make" function. You can make only modified units or build

```
Project  Edit  Search  Window  Program  Macros  Settings
 ListDir.pas              New Window              W
program ListDir;          Activate Next           
Uses crt,dos;             Activate Previous        
                          Centre Window           
const
  month : array [1..1  Window Layout   Arrange All      'Apr';'May';'Jun';
                       Bring to Front  Arrange Views    'Oct';'Nov';'Dec';
var                    Send to Back
  cnt   : byte;                       ✓ Stacked
  x,y   : integer;                      Diagonal
  s     : SearchRec;                    Horizontal   -92 by David Czava)
  ds    : PackedTime;                   Vertical     eed Pascal demo)
  dt    : DateTime;                     Full Size
  protect : array [0..6] of char;
begin
  if ParamCount <> 1 then begin
    writeln('Error - enter a valid directory path');
    halt;
  end;

  FindFirst(ParamStr(1),AnyFile,s);
  FindNext(s);
  while (DosError = 0) do begin
    write(s.Name);                        (write name)
    y := WhereY;                          (save row)
    GotoXY(31,y);                         (tab to column 31)
    if s.Attr and Directory>0 then        (is file a dir?)
      write('Dir':8,' ')
    else
      write(s.Size:8,' ');                (not a dir, so)
                                          (write file size)

    for cnt := 0 to 6 do
      protect[cnt] := '-';                (protection flags)
    if s.Attr and ScriptFlag>0  then protect[0] := 's';
    if s.Attr and PureFlag>0    then protect[1] := 'p';
    if s.Attr and Archive>0     then protect[2] := 'a';
    if s.Attr and ReadFlag>0    then protect[3] := 'r';
    if s.Attr and WriteFlag>0   then protect[4] := 'w';
    if s.Attr and ExecuteFlag>0 then protect[5] := 'e';
    if s.Attr and DeleteFlag>0  then protect[6] := 'd';
    for cnt := 0 to 6 do
      write(protect[cnt]);

    UnPackTime(s.Time,dt);
    write(' ',dt.Day:2,'-',month[dt.Month],'-',dt.Year-1900:2);
    writeln(' ',dt.Hour:2,':',dt.Min:2,':',dt.Sec:2);
    FindNext(s);
  end;
end;
```

```
Find & Replace
 ListDir.pas*  Not found
var   Find & Replace
  cnt
  x,y   Find:      WhereX            Case Sensitive
  s     Change To: WhereY            Whole Words
  ds
  prot  Find Next  Previous  Replace  Replace All  Cancel
begin
  if ParamCount <> 1 then begin
    writeln('Error - enter a valid directory path');
    halt;
  end;

  FindFirst(ParamStr(1),AnyFile,s);
  FindNext(s);
  while (DosError = 0) do begin
    write(s.Name);                        (write name)
    y := WhereY;                          (save row)
    GotoXY(31,y);                         (tab to column 31)
    if s.Attr and Directory>0 then        (is file a dir?)
      write('Dir':8,' ')
    else
      write(s.Size:8,' ');                (not a dir, so)
                                          (write file size)

    for cnt := 0 to 6 do
      protect[cnt] := '-';                (protection flags)
    if s.Attr and ScriptFlag>0  then protect[0] := 's';
    if s.Attr and PureFlag>0    then protect[1] := 'p';
    if s.Attr and Archive>0     then protect[2] := 'a';
    if s.Attr and ReadFlag>0    then protect[3] := 'r';
    if s.Attr and WriteFlag>0   then protect[4] := 'w';
    if s.Attr and ExecuteFlag>0 then protect[5] := 'e';
    if s.Attr and DeleteFlag>0  then protect[6] := 'd';
    for cnt := 0 to 6 do
      write(protect[cnt]);

    UnPackTime(s.Time,dt);
    write(' ',dt.Day:2,'-',month[dt.Month],'-',dt.Year-1900:2);
    writeln(' ',dt.Hour:2,':',dt.Min:2,':',dt.Sec:2);
    FindNext(s);
  end;
end;
```

extensive libraries of pre-compiled units and not worry about linking everything in, resulting in bloated executables.

Full support for single and double IEEE and an extended 10-byte internal math format is provided.

HighSpeed Pascal is *fast*. HiSoft's literature claims a "compilation speed of more than 20,000 lines per minute" and I have found no reason to doubt their claim.

### The Debugger

HiSoft includes the MonAm debugger which is from the Devpac assembler. By selecting "Debug" from the IDE menu or running MonAm from the CLI, a screen with several windows, called a Front Panel, will appear. The windows display the current state of the registers, memory, source code, and disassembled program instructions with symbols. Your program will be suspended at the first instruction with a breakpoint.

> **HighSpeed Pascal is *fast*. HiSoft's literature claims a "compilation speed of more than 20,000 lines per minute" and I have found no reason to doubt their claim.**

You can examine the information in the various windows or issue a command to MonAm. Commands include running, tracing, or single stepping the PC (executing instructions), setting breakpoints, adjusting the display windows, or quitting. All commands are keyboard issued.

There are five different types of breakpoints available including counter and conditional breakpoints. You can set breakpoints by address, expression, or source line number address. There is a help screen which displays pertinent information and lists the breakpoints set, and a history buffer which displays up to five of the most recent breakpoints and exceptions which occurred.

It would take several more pages to list all the commands and features available on MonAm. It suffices to say that it is a well-built and very powerful debugging tool. It is also, in my opinion, a drawback to the entire package. Now, understand that MonAm is a very powerful debugger, but it is also very complex and intimidating—so much so, that I have barely used it.

all units from scratch. Limited conditional compilation is supported. DEFINE and UNDEF provide a means to control range checking, hardware implementations, etc.

Startup and exit code is internal to the compiler, being a part of the System unit. I find this a poor choice as it makes it impossible to modify the startup. Access to the exit code is provided so you can implement termination procedures.

Standard Amiga executables are produced directly by the compiler. There are no provisions for specifying whether data is to be loaded into CHIP memory (for image data). Amiga object files written with Devpak or SAS/C can be linked in and their procedures used with your Pascal code.

HSPC uses "smart linking" to exclude unused procedures from your program. This allows you to create

The rest of the HighSpeed package is geared towards the casual or professional programmer who desires the simplicity of an IDE with a powerful yet clean and understandable version of Pascal. Yet this debugger is for programmers with extensive knowledge of assembler and low-level debugging skills.

To be fair, I have used the debugger successfully without much hair-pulling, but it is certainly not what I would call "user friendly." HighSpeed Pascal virtually begs for a good source level debugger along the lines of the *Benchmark* or *Manx SLD*.

### The Manuals

HighSpeed Pascal comes with two lay-flat, spiral-bound manuals. The first is a 200-page User Manual which includes installation, a quickie tutorial, separate chapters on the editor, compiler, and debugger, usage of included utilities, and a description of all the Amiga Units that are included for Amiga system access.

The second manual is the Technical Reference manual which describes the Pascal implementation and the other units supplied, including the System, CRT, DOS, and Graph TurboPascal units.

Each procedure and function is detailed with the calling declaration, a function description, comments about the procedure, references to other or similar functions, and a standalone example program which uses the procedure.

Both manuals have a Table of Contents as well as moderately useful indexes. I would have liked a reference card of all the many functions. There are just too many to remember, and paging through the 280+ page reference manual is taxing.

The manuals are not Pascal language tutorials. They are references to the HighSpeed Pascal implementation and nothing more. If you do not know Pascal, buy a TurboPascal tutorial book and you will be ready to go.

### Conclusions

I have been programming in Modula 2 for many years, but have never tried Pascal. I picked up the language very quickly just by examining public domain TurboPascal source code. The languages are all very similar and the units concept was very familiar to me.

I am much impressed with the speed and operation of HSPascal. The IDE makes working with HSPascal quick and pleasurable. I long for a better source level debugger, though.

The TurboPascal compatibility is what really makes this package shine. I have a lot of TurboPascal source code which I acquired from various public domain libraries and much of it compiles and runs without change. If you are about to start porting that special TP program into C so it can be used on the Amiga, this package will pay for itself in moments.

I have not encountered any compiler bugs. I did note a minor problem in a System procedure, which HiSoft said would be corrected in the next upgrade. There may be other problems, but after several weeks of heavy use, I haven't found them yet.

The editor's method of highlighting text is somewhat awkward, but certainly not buggy. The manuals are informative with many examples and few typographical errors.

Several levels of product and technical support are

available from HiSoft; however, as the company is based in the U.K., the support programs are not very economical for U.S. owners. HiSoft representatives offer online support with regular appearances on both GEnie and Bix. They have been very prompt and helpful with my many questions. This is the alternative language I've been waiting for. HighSpeed Pascal is for the rest of us. C you later...

TurboPascal is a registered trademark of Borland Inc. Amiga is a registered trademark of Commodore-Amiga Inc. HighSpeed Pascal and MonAm are registered trademarks of HiSoft. GEnie is a registered trademark of General Electric. Bix is a registered trademark of General Videotex Corp.

☑

Please write to:
**David Czaya**
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

# Understanding the Console Device

*by David A. Blackwell*

## Introduction

Some time ago I attended a computer programming course at a local junior college. The computers we used for the course were IBM compatibles. Being mostly familiar with the Amiga, I thought this would be an exciting learning experience. I was somewhat disappointed when I found out that the IBM machines were not nearly as challenging to program as the Amiga. Some people may enjoy the ease of such an uncomplicated operating system. On the other hand, I enjoy the challenge of programming an operating system as powerful and sophisticated as the one in the Amiga.

I began to try to duplicate the programs we were writing in my college course on my Amiga. Some of the aspects of these programs included manipulating the cursor on the screen and performing single character input. The standard C routines provided in my compiler did not have the capability I was looking for. There were no functions at all to handle the cursor. The character input routines required me to press the return key before my program would receive the character. I wanted to be able to read the key as it was pressed. I decided my only option was to search through the *Amiga ROM Kernal Reference Manuals* to see what I could find.

## Console Device

The console device seemed to be what I needed to make my program work the way I wanted. The information in the reference manual, although complete, did not clearly convey to me exactly what I wanted to know. I still had nagging questions about certain points I wasn't sure about, so I did what I always do under those circumstances: I wrote a program to experiment with the console device and gain firsthand experience with it. It is that experience that I hope to pass on to you in this article and give you a basic understanding of the console device that will allow you to quickly put the console device to use. I suggest you read this article in conjunction with the *Amiga ROM Kernal Reference Manuals*. If you do not own these reference manuals, I highly recommend them.

## Standard Device Commands

The system programmers for the Amiga operating system chose eight standard device commands for all devices designed to operate on the Amiga. Devices do not exactly have to support every command as long as they respond to each command. Normally an error code is returned for the commands a device does not support.

## Console Device Commands

Of the eight standard device commands, the console device only supports three. They are the CMD_CLEAR, CMD_READ and CMD_WRITE commands. The fact that only three standard commands are supported does not detract at all from the console device, and actually makes it quite easy to use. The CMD_CLEAR command instructs the console device to clear its display. The CMD_READ command causes the console device to read an indicated number of characters from the input stream and return them to you. Finally, the CMD_WRITE command makes the console device write the provided string to its display. The console device has additional non-standard commands but they are beyond the scope of this article. After you have decided that you want to use the console device, you must perform the following steps to set it up for use with your intuition window.

## Getting Started

The console device, with only one exception, must be attached to an already opened window. (The one exception to this requirement is beyond the scope of this article. Maybe I will be able to cover it in a future article.) Once you have the window open you are ready to continue.

The next step is to allocate an I/O request structure to use with the console device. This is extremely easy to do and can be accomplished with the following code:

```
struct IOStdReq *myMessage;

myMessage = CreateStdIO(myPort);
```

If you are going to use synchronous I/O, you will want to allocate two structures for use. You will use one for your read requests and the other for your write commands. If you want to use asynchronous I/O you can get by with one global structure and clone all your read and write requests from it as you allocate them. This is the method I use in my demonstration program. I feel it is the most flexible since it allows you to accept input from more than a single source. In my demonstration program you will notice that I use both the console device and an IDCMP (Intuition's Direct Communication Message Port) for input.

Once you have an I/O request structure allocated, place the pointer to your window in its io_Data field and the length of the window structure in its io_Length field. Then you call

# • Standard Device Commands
# • Console Input
# • Console Output
# • Getting Started

the OpenDevice() command. The following code demonstrates these points.

```
myMessage->io_Data = (APTR) Window;
myMessage->io_Length = sizeof(struct Window);
OpenDevice("console.device", 0, myMessage, 0);
```

The OpenDevice() function fills in the io_Device and io_Unit fields of your I/O request structure. In synchronous communications, the io_Device and io_Unit fields are the ones you will duplicate in your second I/O request structure. In asynchronous communications, you clone the same two fields in each I/O request structure you send to the console device. Once all this is done, you are ready to start communicating with the console device.

## Console I/O

If you are using synchronous communications, most of your work is already done. To write to the console device, you need only place a pointer to the string you want written and its length in the io_Data and io_Length fields of your I/O request structure. To read from the console device, you put a pointer to a data buffer and its length in the io_Data and io_Length fields of your I/O request structure. The io_Length field not only specifies the length of your data buffer but also tells the console device the maximum number of characters to return. Asynchronous communications on the other hand are a bit more involved.

To perform asynchronous communications, you will need to allocate a new I/O request structure each time you wish to communicate with the console device. You clone the required fields from your global I/O request structure into the new I/O request structure you just allocated. Next, place the console command in your structure's io_Command field. Now allocate the data buffer memory area and place its pointer and length into the io_Data and io_Length fields. If your data area has already been allocated, all you need to do is put the proper values in the correct structure fields. That is the method I use in my demonstration program. An example of functions to handle asynchronous read and write commands follow:

```
void ConsoleRead(UBYTE *string, int length)

struct IOStdReq *msg;

if ( !(msg = CreateStdIO(myPort)) )
```

```
puts("Unable to read from the console.");
goto cr_exit;

msg->io_Device = myMessage->io_Device; /* Clone these */
msg->io_Unit = myMessage->io_Unit; /* fields */
msg->io_Command = CMD_READ;
msg->io_Data = (APTR)string;
msg->io_Length = length;

SendIO(msg);

cr_exit:
return;


void ConsoleWrite(UBYTE *string, int length)

struct IOStdReq *msg;

if ( !(msg = CreateStdIO(myPort)) )

puts("Unable to write to the console.");
goto cw_exit;

msg->io_Device = myMessage->io_Device; /* Clone these */
msg->io_Unit = myMessage->io_Unit; /* fields */
msg->io_Command = CMD_WRITE;
msg->io_Data = (APTR)string;
msg->io_Length = length;

SendIO(msg);

cw_exit:
return;
```

You will notice that I used the SendIO() function in my asynchronous communication functions. If you are using synchronous communications, you will want to use the DoIO() function since it waits for the device to finish its processing before it returns control back to your program. That is how you send messages back and forth between your program and the console device. Now let's look and what you can send or receive.

## Console Output

Besides the usual character strings you can send to the console device, there are numerous control sequences, both ANSI standard and Amiga non-ANSI standard, which you can send to the console device to manipulate it in various ways. Lists One and Two contain a complete list of these control sequences.

Where it is indicated that you need to enter a value, you always use ASCII characters to represent the digits of the numeric value. For example, to represent the numeric value 20 in ASCII characters, you would use the hexadecimal numbers 32 and 30. The reason you use hexadecimal numbers rather than decimal is that the hexadecimal number 30 represents zero. Therefore, all you need to do to get the digit you want is to add its value to the hexadecimal number 30. To get the number five just add 5 to 30 to get the hexadecimal number 35. I'm sure you will agree that the hexadecimal number 30 to represent zero is easier to read and understand than the decimal number 48:

Concerning List Two, you can enter more than one event type into the SET RAW EVENTS and RESET RAW EVENTS control sequences as long as they a separated by semicolons (hex value 3B). List Three contains a listing of all the event types used in these control sequences. These event types determine what type of input you receive from the console device.

## Console Input

If you have not selected RAW input events, then you will receive the ASCII-equivalent character for the ANSI standard keys on the keyboard. For the other keys, you will receive an escape sequence of two to four characters. List Four contains the escape sequences for the non-ANSI standard keys. This form of input, sometimes called COOKED KEY EVENTS, is the default. This is the easiest to receive and process. The one disadvantage is that you do not receive any other information on the key press other than shifted or unshifted. You don't know if the Alt, AMIGA or Ctrl keys were used.

If you need more information about the keyboard input events, you send the SET RAW EVENTS control sequence to the console device. You can request any combination of input events from List Three. If later you want to reduce the number of input events you want to receive, you issue the RESET RAW EVENTS control sequence with the appropriate event number(s) that you no longer want. With these two control sequences you can precisely control the information you receive from the console device. The format of the returned information is very different from the cooked format and requires more processing to use fully.

Instead of receiving a standard ANSI-standard character code you receive what is known as a "complex input event report." This report takes the form of:

```
<CSI>class;subclass;keycode;qualifiers;x;y;seconds;microseconds
```

The exact format of this report is what was the biggest problem for me as I was studying the *Amiga ROM Kernal Manual*. In experimenting with the console device, I discovered that this is returned as a null-terminated ASCII string. To use the values in the report, you must convert the different parts of the string you want to use to the correct numeric format. The first five fields of the complex input event report as broken out as follows:

```
<CSI> - one-byte field (hex 9B)
class - RAW input event type
subclass - usually 0, 1 if mouse is in right
controller.
keycode - raw key number or mouse information
qualifiers - indicates the state of the
keyboard and system
```

The remainder of the fields in the complex input event report are self-explanatory. It is important to remember that the keycode is not the ASCII code but instead, it is the number of the key in the keyboard matrix. To convert it to the ASCII character code, you call the RawKeyConvert() function. This function requires a global variable by the name of "ConsoleDevice" be assigned

# List One:

## ANSI Standard Control Sequences

| | |
|---|---|
| BACKSPACE | 08 |
| LINE FEED | 0A |
| VERTICAL TAB | 0B |
| FORM FEED | 0C |
| CARRIAGE RETURN | 0D |
| SHIFT IN | 0E |
| SHIFT OUT | OF |
| ESC | 1B |
| CSI (Control Sequence Introducer) | 9B |
| RESET TO INITIAL STATE | 1B 63 |
| INSERT [N] CHARACTERS | 9B [N] 40 |
| CURSOR UP [N] LINES | 9B [N] 41 |
| CURSOR DOWN [N] LINES | 9B [N] 42 |
| CURSOR FORWARD [N] SPACES | 9B [N] 43 |
| CURSOR BACKWARD [N] SPACES | 9B [N] 44 |
| CURSOR NEXT LINE [N] (to column 1) | 9B [N] 45 |
| CURSOR PRECEDING LINE [N] (col. 1) | 9B [N] 46 |
| MOVE CURSOR TO ROW; COLUMN | 9B [R] [3B C] 48 |
| ERASE TO END OF DISPLAY | 9B 4A |
| ERASE TO END OF LINE | 9B 4B |
| INSERT LINE | 9B 4C |
| DELETE LINE | 9B 4D |
| DELETE CHARACTER [N] | 9B [N] 50 |
| SCROLL UP [N] LINES | 9B [N] 53 |
| SCROLL DOWN [N] LINES | 9B [N] 54 |
| SET LINEFEED MODE (RETURN-LINEFEED) | 9B 32 30 68 |
| SET LINEFEED MODE (LINEFEED ONLY) | 9B 32 30 6C |
| DEVICE STATUS REPORT | 9B 36 6E |

the pointer to the console device structure. You can do this after you call the OpenDevice() function as demonstrated in the following code:

```
    OpenDevice("console.device", 0L,
myMessage, 0L);
    ConsoleDevice = myMessage-
>io_Device;
```

Then before you call the RawKeyConvert() function, you allocate an input event structure and fill in the ie_Code and ie_Qualifier fields with the keycode and qualifier values returned in the complex input event report. However, you can't use the values as they are returned in the report. You need to convert them to a numeric value. There are two functions in my demonstration program that provide one method to do this. After the input event structure is filled in, then call the RawKeyConvert() function. One of the arguments you supply to this function is a buffer for the converted character. When this function call returns, this is where you will find your converted character. The difference between using this method to get your ASCII character code and just asking for cooked keys in the first place is the reporting of qualifiers in the complex input event report.

The qualifiers returned are quite explicit. Not only does it report if the SHIFT, Alt or AMIGA keys are being held down, but it also distinguishes between the left and right keys. There are many other qualifiers that you can receive also. The qualifiers you receive are determined by which class of raw key events you requested to receive. List Five contains a complete list of the possible qualifiers. Some of these qualifiers you probably will never need. I included a function in my demonstration program that converts the qualifier section of the complex input event report to a numeric value that can easily be tested and I also demonstrate how to test it for several of the qualifiers.

### Demonstration Program

The accompanying demonstration program should help you more fully understand how you can use the console device for you own needs and how to use both the cooked key and raw key return events. I only request raw keyboard and mouse events in my program, but it could be easily modified to experiment with any of the raw event types you are interested in. I have also shown an easy way to make macros that are easy to understand and use in your own program. These macros provide an easy way to home the cursor, clear the screen, move the cursor to any position on the screen, etc. The program runs with raw key events selected as its default. With the gadgets, you can toggle between cooked and raw key events. The program waits for keyboard or mouse input and displays the return value from the console device to the screen.

## List Two:
### *Amiga Console-Control Sequences*

| | |
|---|---|
| ENABLE SCROLL (default) | 9B 3E 31 68 |
| DISABLE SCROLL | 9B 3E 31 6C |
| AUTOWRAP ON (default) | 9B 3F 37 68 |
| AUTOWRAP OFF | 9B 3F 37 6C |
| SET PAGE LENGTH | 9B <length> 74 |
| SET LINE LENGTH | 9B <width> 75 |
| SET LEFT OFFSET | 9B <offset> 78 |
| SET TOP OFFSET | 9B <offset> 79 |
| SET RAW EVENTS | 9B <event types> 7B |
| RESET RAW EVENTS | 9B <event types> 7D |
| SET CURSOR RENDITION | |
| Invisible: | 9B 30 20 70 |
| Visible: | 9B 20 70 |
| WINDOW STATUS REQUEST | 9B 30 20 71 |

## List Three:
### *RAW Event Types*

| | |
|---|---|
| 0 | No-op |
| 1 | RAW keyboard input |
| 2 | RAW mouse input |
| 3 | Window Activated Event |
| 4 | Pointer position |
| 5 | (unused) |
| 6 | Timer |
| 7 | Gadget pressed |
| 8 | Gadget released |
| 9 | Requester activity |
| 10 | Menu numbers |
| 11 | Close gadget |
| 12 | Window resized |
| 13 | Window refreshed |
| 14 | Preferences changed |
| 15 | Disk removed |
| 16 | Disk inserted |

For raw key events, it also shows the converted raw key and the most basic qualifiers that are pressed.

Experiment with this program to get the most out of it you can. As always, I can be reached for questions or comments on GEnie using the e-mail address, D.Blackwell1.

## List Four:

*non-ANSI Standard Escape Sequences*

| Key | Unshifted | Shifted |
|-----|-----------|---------|
| F1 | <CSI>0~ | <CSI>10~ |
| F2 | <CSI>1~ | <CSI>11~ |
| F3 | <CSI>2~ | <CSI>12~ |
| F4 | <CSI>3~ | <CSI>13~ |
| F5 | <CSI>4~ | <CSI>14~ |
| F6 | <CSI>5~ | <CSI>15~ |
| F7 | <CSI>6~ | <CSI>16~ |
| F8 | <CSI>7~ | <CSI>17~ |
| F9 | <CSI>8~ | <CSI>18~ |
| F10 | <CSI>9~ | <CSI>19~ |
| HELP | <CSI>?~ | <CSI>?~ |

Arrow keys:

| | | |
|-----|-----------|---------|
| UP | <CSI>A | <CSI>T |
| DOWN | <CSI>B | <CSI>S |
| LEFT | <CSI>D | <CSI> A |
| RIGHT | <CSI>C | <CSI> @ |

(notice the space in last two shifted arrow keys sequences)
(<CSI> is equivilant to 9B)

## List Five:

*Input Event Qualifiers*

| | |
|-----|-----|
| Left Shift | Numeric pad |
| Right Shift | Repeat |
| Caps Lock | Interrupt |
| Ctrl | Multi-boradcast |
| Left Alt | Left Mouse Button |
| Right Alt | Right Mouse Button |
| Left Amiga key | Middle Mouse Button |
| Right Amiga key | Relative Mouse |

## Source Code

```
ShowConsole.c
A program to help you gain experience with the console
device.

        Manx Aztec C 5.2a source

        cc ShowConsole.c
        ln ShowConsole.o s.lib c.lib

*/

#include <stdio.h>
#include <string.h>
#include <exec/types.h>
#include <exec/io.h>
#include <clib/console_protos.h>
#include <intuition/intuition.h>
#include "ShowConsoleGadgets.c"

#define MEMF_CLEAR (1<<16)

UBYTE ERASEEOL[] = { 0x9b, 'K' };
UBYTE ERASEEOD[] = { 0x9b, 'J' };
UBYTE HOME[]    = { 0x9b, '1', ';', '1', 'H' };
UBYTE MOVE[]    = { 0x9b, '0', '0', ';', '0', '0', 'H'
};
UBYTE MOVE2[]   = { 0x9b, '2', '1', ';', '1', '6', 'H'
};
UBYTE ONEDOWN[] = { 0x9b, '2', ';', '1', 'H' };
UBYTE RAWKEYINPUT[] = { 0x9b, '1', ';', '2', '{' };
UBYTE COOKEDKEYS[]     = { 0x9b, '1', ';', '2', '}' };
UBYTE CURSORON[] = { 0x9b, 0x20, 0x70 };
UBYTE CURSOROFF[]      = { 0x9b, 0x30, 0x20, 0x70 };

char *mask[] = { "struct IOStdReq {\n",
    "       struct Node *(........) (ln_Succ)\n",
    "       struct Node *(........) (ln_Pred)\n",
    "       UBYTE ...       (ln_Type)\n",
    "       BYTE    ...     (ln_Pri)\n",
    "       char *(........) (ln_Name)\n",
    "       struct MsgPort *(........)
(mn_ReplyPort)\n",
    "       UWORD .....     (mn_Length)\n",
    "       struct Device *(........)
(io_Device)\n",
    "       struct Unit *(........) (io_Unit)\n",
    "       UWORD ..... (io_Command)\n",
    "       UBYTE ...       (io_Flags)\n",
    "       BYTE    ...     (io_Error)\n",
    "       ULONG ........ (io_Actual)\n",
    "       ULONG ........ (io_Length)\n",
    "       APTR    ........ (io_Data)\n",
    "       ULONG ........ (io_Offset)\n",
    "};     ",
    "<CSI>class;subclass;keycode;qualifiers;....|\n"
};

char *quals[] = { "Left Shift",
```

```
                "Right Shift",
                "Ctrl",
                "Left Alt",
                "Right Alt",
                "Left Amiga",
                "Right Amiga"
                );

            /* Defines */
    #define IB          IntuitionBase
    #define BufferLen        121
    #define ERASE_EOL()         ConsoleWrite(ERASEEOL,2)
    #define ERASE_EOD()         ConsoleWrite(ERASEEOD,2)
    #define HOME_CURSOR()       ConsoleWrite(HOME,5)
    #define MOVE_CURSOR()       ConsoleWrite(MOVE,7)
    #define ONEDOWN_CURSOR() ConsoleWrite(ONEDOWN,5)
    #define DO_RETURNEDDATA()       ConsoleWrite(MOVE2,7)
    #define GET_RAWKEYINPUT()
    ConsoleWrite(RAWKEYINPUT,5)
    #define GET_COOKEDKEYS() ConsoleWrite(COOKEDKEYS,5)
    #define SET_CURSORON()   ConsoleWrite(CURSORON,3)
    #define SET_CURSOROFF()  ConsoleWrite(CURSOROFF,4)

            /* Prototypes */
    extern struct MsgPort *CreatePort(char *, long);
    extern struct IOStdReq *CreateStdIO(struct MsgPort *);
    extern struct Window *OpenWindow(struct NewWindow *);
    extern void *OpenLibrary(char *, long);
    extern long OpenDevice(char *, long, struct IORequest *,
    long);
    void shutdown(void);
    void ConsoleWrite(UBYTE *, int);
    void ConsoleRead(UBYTE *, int);
    void PrintCharacter(UBYTE *);
    void ClearCharacter(void);
    void PrintQualifiers(UBYTE *);
    void ClearQualifiers(void);
    void PrintConsoleInfo(struct IOStdReq *);
    void ClearScrn(void);
    struct Window *DoWindow(void);
    UWORD convert_code(UBYTE *string);
    UWORD convert_qualifiers(UBYTE *string);

    void ubtoa(UBYTE, char *);
    void uwtoa(UWORD, char *);
    void ultoh(ULONG, char *);

            /* Pragmas */
    #pragma regcall(ubtoa(d0,a0))
    #pragma regcall(uwtoa(d0,a0))
    #pragma regcall(ultoh(d0,a0))

            /* Globals */
    struct Device *ConsoleDevice;
    struct IntuitionBase *IntuitionBase;
    struct MsgPort *myPort;
    struct IOStdReq *myMessage;
    struct Window *Window;
    UBYTE buffer[BufferLen+1];
    BOOL raw_keys = TRUE;

    void main(void)
    {
            UBYTE key;
            LONG gadget_class;
            int x;
            long error, windowsignal, consolesignal,
    returnedsignals;
            struct Gadget *gadget_selected;
            struct IntuiMessage *my_message;
            struct IOStdReq *my_ioreq;
            BOOL done = FALSE;

            IntuitionBase = (struct IB
    *)OpenLibrary("intuition.library", 0L);

            if ( !(Window = DoWindow()) )
            {
            puts("Not enough memory to open a window.");
            goto main_exit;
            }

            if ( !(myPort = CreatePort("myConPort", 0L)) )
            {
            puts("Not enough memory for a message port.");
            goto main_exit;
            }

            if ( !(myMessage = CreateStdIO(myPort)) )
            {
            puts("Not enough memory for a message struc-
    ture.");
            goto main_exit;
            }

            myMessage->io_Data = (APTR) Window;
            myMessage->io_Length = sizeof(struct Window);
            error=OpenDevice("console.device", 0L, (struct
    IORequest *)myMessage, 0L);
            ConsoleDevice = myMessage->io_Device;

            if ( error )
            {
            puts("Problem encountered opening console
    device.");
            goto main_exit;
            }

            windowsignal = 1 << Window->UserPort->mp_SigBit;
            consolesignal = 1 << myPort->mp_SigBit;

            GET_RAWKEYINPUT(); /* The default is to request
    raw key events. */
            ClearScrn();
            RefreshGList(&GadgetList1, Window, 0L, -1);

            ConsoleWrite((UBYTE *)"ShowConsole by David A.
    Blackwell", -1);
            PrintConsoleInfo(myMessage);

            ConsoleRead(buffer, BufferLen-1);

            SET_CURSOROFF();

            while ( !done )
            {
            returnedsignals =
    Wait(windowsignal|consolesignal);
            if ( returnedsignals & windowsignal )
            {
            while (my_message = (struct IntuiMessage
    *)GetMsg(Window->UserPort))
            {
            gadget_class = my_message->Class;
```

```
        gadget_selected = (struct Gadget *)my_message-
>IAddress;
        ReplyMsg((struct Message *)my_message);
        switch(gadget_class)
        {
        case GADGETUP:
        switch (gadget_selected->GadgetID)
        {
        case RAW_KEYS:
        if ( !raw_keys )
        {
        GET_RAWKEYINPUT();
        raw_keys = TRUE;
        }
        break;
        case QUIT:
        done = TRUE;
        break;
        case COOKED_KEYS:
        if ( raw_keys )
        {
        GET_COOKEDKEYS();
        raw_keys = FALSE;
        }
        break;
        default:
        puts("Unexpected gadget selection.");
        }
        break;
        default:
        puts("Unexpected Intuition event.");
        }
        }
        }
        if ( returnedsignals & consolesignal )
        {
        while ( (my_ioreq = (struct IOStdReq
*)GetMsg(myPort)) )
        {
        if ( my_ioreq->io_Command == CMD_READ )
        {
        PrintConsoleInfo(my_ioreq);
        ConsoleRead(buffer, BufferLen-1);
        }
        DeleteStdIO(my_ioreq);
        }
        }

        SET_CURSORON();

        CloseDevice(myMessage);

main_exit:
        shutdown();
}

void shutdown(void)
{
        struct IOStdReq *my_ioreq;

        if ( myMessage )
        DeleteStdIO(myMessage);

        if ( myPort )
        {
        while ( (my_ioreq = (struct IOStdReq
```

```
*)GetMsg(myPort)) )
        {
        DeleteStdIO(my_ioreq);
        }
        DeletePort(myPort);
        }

        if ( Window )
        CloseWindow(Window);

        if ( IntuitionBase )
        CloseLibrary(IntuitionBase);

}

/* DoWindow - open a window and return a pointer to the
window structure. */

struct Window *DoWindow(void)
{
        struct NewWindow *nw;
        struct Window *window = (struct Window *) 0;

        if ( !(nw = (struct NewWindow
*)AllocMem(sizeof(struct NewWindow), MEMF_CLEAR)) )
        goto dw_exit;

        nw->LeftEdge = 0;
        nw->TopEdge = 0;
        nw->Width = 640;
        nw->Height = 200;
        nw->DetailPen = 0;
        nw->BlockPen = 1;
        nw->Title = (UBYTE *) "ShowConsole Window";
        nw->Flags = ACTIVATE|WINDOWDEPTH|SMART_REFRESH;
        nw->IDCMPFlags = GADGETUP;
        nw->Type = WBENCHSCREEN;
        nw->FirstGadget = &GadgetList1;
        nw->CheckMark = 0;
        nw->Screen = 0;
        nw->BitMap = 0;
        nw->MinWidth = 0;
        nw->MinHeight = 0;
        nw->MaxWidth = 640;
        nw->MaxHeight = 200;

        window = OpenWindow(nw);

dw_exit:

        return(window);
}

/* ConsoleWrite - Write a string to the console device.
*/

void ConsoleWrite(UBYTE *string, int length)
{
        struct IOStdReq *msg;

        if ( !(msg = CreateStdIO(myPort)) )
        {
        puts("Unable to write to the console.");
        goto cw_exit;
        }

        msg->io_Device = myMessage->io_Device;
```

```c
        msg->io_Unit = myMessage->io_Unit;
        msg->io_Command = CMD_WRITE;
        msg->io_Data = (APTR)string;
        msg->io_Length = length;

        SendIO(msg);

cw_exit:
        return;
}

/* ConsoleRead - Read a given number of characters from
the console device. */

void ConsoleRead(UBYTE *string, int length)
{
        struct IOStdReq *msg;

        if ( !(msg = CreateStdIO(myPort)) )
        {
        puts("Unable to read from the console.");
        goto cr_exit;
        }

        msg->io_Device = myMessage->io_Device;
        msg->io_Unit = myMessage->io_Unit;
        msg->io_Command = CMD_READ;
        msg->io_Data = (APTR)string;
        msg->io_Length = length;

        SendIO(msg);

cr_exit:
        return;
}

/* ClearScrn - Home the cursor and clear the screen. */

void ClearScrn(void)
{
        HOME_CURSOR();
        ERASE_EOD();
}

void PrintConsoleInfo(struct IOStdReq *msg)
{
        int x;
        static UBYTE tempb[4] = { "ESC " };

        ultoh((ULONG)msg->io_Message.mn_Node.ln_Succ,
mask[1]+17);
        ultoh((ULONG)msg->io_Message.mn_Node.ln_Pred,
mask[2]+17);
        ubtoa((UBYTE)msg->io_Message.mn_Node.ln_Type,
mask[3]+9);
        ubtoa((UBYTE)msg->io_Message.mn_Node.ln_Pri,
mask[4]+9);
        ultoh((ULONG)msg->io_Message.mn_Node.ln_Name,
mask[5]+10);
        ultoh((ULONG)msg->io_Message.mn_ReplyPort,
mask[6]+20);
        uwtoa((UWORD)msg->io_Message.mn_Length,
mask[7]+9);
        ultoh((ULONG)msg->io_Device, mask[8]+19);
        ultoh((ULONG)msg->io_Unit, mask[9]+17);
        uwtoa((UWORD)msg->io_Command, mask[10]+9);
        ubtoa((UBYTE)msg->io_Flags, mask[11]+9);

        ubtoa((UBYTE)msg->io_Error, mask[12]+9);
        ultoh((ULONG)msg->io_Actual, mask[13]+9);
        ultoh((ULONG)msg->io_Length, mask[14]+9);
        ultoh((ULONG)msg->io_Data, mask[15]+9);
        ultoh((ULONG)msg->io_Offset, mask[16]+9);

        ONEDOWN_CURSOR();

        for ( x = 0 ; x < 19 ; x++ )
        ConsoleWrite((UBYTE *)mask[x], -1);

        ConsoleWrite((UBYTE *)"Returned Data: ", -1);
        MOVE[1] = '0';
        MOVE[2] = '2';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ConsoleWrite((UBYTE *)"Char: ", -1);
        MOVE[1] = '0';
        MOVE[2] = '4';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ConsoleWrite((UBYTE *)"Qualifiers: ", -1);
        DO_RETURNEDDATA();
        ERASE_EOL();
        if ( raw_keys )
        {
        if ( buffer[0] )
        {
        ConsoleWrite(tempb, 4);
        ConsoleWrite(&buffer[1], -1);
        PrintCharacter(buffer);
        PrintQualifiers(buffer);
        }
        }else{
        ConsoleWrite(&buffer[0], -1);
        ClearCharacter();
        ClearQualifiers();
        }

        for( x = 0 ; buffer[x] != 0 ; x++ )
        buffer[x] = 0;
        if ( MOVE2[2] == '0' )
        {
        MOVE2[2] = '1';
        }else{
        MOVE2[2] = '0';
        }

}

void PrintCharacter(UBYTE *characterbuffer)
{
        char convertbuffer[4];
        long numberchars;
        static struct InputEvent ievent = {NULL,
IECLASS_RAWKEY, 0, 0, 0};

        ievent.ie_Code = convert_code(characterbuffer);
        ievent.ie_Qualifier =
convert_qualifiers(characterbuffer);
        ievent.ie_position.ie_addr = NULL;

        numberchars = RawKeyConvert(&ievent, (STRPTR)
convertbuffer, 4L, 0L);
```

```
        if (!numberchars)                                        MOVE[1] = '0';
        goto pc_exit;                                            MOVE[2] = '7';
                                                                 MOVE[4] = '5';
        MOVE[1] = '0';                                           MOVE[5] = '5';
        MOVE[2] = '2';                                           MOVE_CURSOR();
        MOVE[4] = '6';                                           ConsoleWrite((UBYTE *)quals[2], -1);
        MOVE[5] = '1';                                           }else{
        MOVE_CURSOR();                                           MOVE[1] = '0';
        ConsoleWrite((UBYTE *)convertbuffer, 1);                 MOVE[2] = '7';
                                                                 MOVE[4] = '5';
pc_exit:                                                         MOVE[5] = '5';
        return;                                                  MOVE_CURSOR();
}                                                                ERASE_EOL();
                                                                 }
void ClearCharacter(void)
{                                                            if (char_qualifiers & IEQUALIFIER_LALT)
        MOVE[1] = '0';                                           {
        MOVE[2] = '2';                                           MOVE[1] = '0';
        MOVE[4] = '6';                                           MOVE[2] = '8';
        MOVE[5] = '1';                                           MOVE[4] = '5';
        MOVE_CURSOR();                                           MOVE[5] = '5';
        ConsoleWrite((UBYTE *)" ", 1);                           MOVE_CURSOR();
                                                                 ConsoleWrite((UBYTE *)quals[3], -1);
}                                                                }else{
                                                                 MOVE[1] = '0';
void PrintQualifiers(UBYTE *characterbuffer)                     MOVE[2] = '8';
{                                                                MOVE[4] = '5';
        UWORD char_qualifiers;                                   MOVE[5] = '5';
                                                                 MOVE_CURSOR();
        char_qualifiers =                                        ERASE_EOL();
convert_qualifiers(characterbuffer);                             }

        if (char_qualifiers & IEQUALIFIER_LSHIFT)            if (char_qualifiers & IEQUALIFIER_RALT)
        {                                                        {
        MOVE[1] = '0';                                           MOVE[1] = '0';
        MOVE[2] = '5';                                           MOVE[2] = '9';
        MOVE[4] = '5';                                           MOVE[4] = '5';
        MOVE[5] = '5';                                           MOVE[5] = '5';
        MOVE_CURSOR();                                           MOVE_CURSOR();
        ConsoleWrite((UBYTE *)quals[0], -1);                     ConsoleWrite((UBYTE *)quals[4], -1);
        }else{                                                   }else{
        MOVE[1] = '0';                                           MOVE[1] = '0';
        MOVE[2] = '5';                                           MOVE[2] = '9';
        MOVE[4] = '5';                                           MOVE[4] = '5';
        MOVE[5] = '5';                                           MOVE[5] = '5';
        MOVE_CURSOR();                                           MOVE_CURSOR();
        ERASE_EOL();                                             ERASE_EOL();
        }                                                        }

        if (char_qualifiers & IEQUALIFIER_RSHIFT)            if (char_qualifiers & IEQUALIFIER_LCOMMAND)
        {                                                        {
        MOVE[1] = '0';                                           MOVE[1] = '1';
        MOVE[2] = '6';                                           MOVE[2] = '0';
        MOVE[4] = '5';                                           MOVE[4] = '5';
        MOVE[5] = '5';                                           MOVE[5] = '5';
        MOVE_CURSOR();                                           MOVE_CURSOR();
        ConsoleWrite((UBYTE *)quals[1], -1);                     ConsoleWrite((UBYTE *)quals[5], -1);
        }else{                                                   }else{
        MOVE[1] = '0';                                           MOVE[1] = '1';
        MOVE[2] = '6';                                           MOVE[2] = '0';
        MOVE[4] = '5';                                           MOVE[4] = '5';
        MOVE[5] = '5';                                           MOVE[5] = '5';
        MOVE_CURSOR();                                           MOVE_CURSOR();
        ERASE_EOL();                                             ERASE_EOL();
        }                                                        }

        if (char_qualifiers & IEQUALIFIER_CONTROL)          if (char_qualifiers & IEQUALIFIER_RCOMMAND)
        {
```

```c
        {
        MOVE[1] = '1';
        MOVE[2] = '1';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ConsoleWrite((UBYTE *)quals[6], -1);
        }else{
        MOVE[1] = '1';
        MOVE[2] = '1';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ERASE_EOL();
        }

}

void ClearQualifiers(void)
{

        MOVE[1] = '0';
        MOVE[2] = '5';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ERASE_EOL();

        MOVE[1] = '0';
        MOVE[2] = '6';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ERASE_EOL();

        MOVE[1] = '0';
        MOVE[2] = '7';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ERASE_EOL();

        MOVE[1] = '0';
        MOVE[2] = '8';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ERASE_EOL();

        MOVE[1] = '0';
        MOVE[2] = '9';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ERASE_EOL();

        MOVE[1] = '1';
        MOVE[2] = '0';
        MOVE[4] = '5';
        MOVE[5] = '5';
        MOVE_CURSOR();
        ERASE_EOL();

        MOVE[1] = '1';
        MOVE[2] = '1';
        MOVE[4] = '5';
        MOVE[5] = '5';
```

```c
        MOVE_CURSOR();
        ERASE_EOL();

}

UWORD convert_code(UBYTE *string)
{
        int x;
        char semicolon = ';';
        UBYTE temp;
        UBYTE *begining, *ending;
        UWORD rtnval = 0;

        for ( x = 0, begining = string ; x < 2 ; x++ )
        {
        begining++;
        begining = (UBYTE *)strchr((char *)begining,
(int)semicolon);
        if ( !begining )
        goto cc_exit;
        }

        begining++;
        ending = (UBYTE *)strchr((char *)begining,
(int)semicolon);

        if ( !ending )
        goto cc_exit;

        temp = *ending;
        *ending = (UBYTE) 0;
        rtnval = (UWORD) atol((char *)begining);
        *ending = temp;

cc_exit:
        return(rtnval);
}

UWORD convert_qualifiers(UBYTE *string)
{
        int x;
        char semicolon = ';';
        UBYTE *begining, *ending;
        UBYTE temp;
        UWORD rtnval = 0;

        for ( x = 0, begining = string ; x < 3 ; x++ )
        {
        begining++;
        begining = (UBYTE *)strchr((char *)begining,
(int)semicolon);
        if ( !begining )
        goto cq_exit;
        }

        begining++;
        ending = (UBYTE *)strchr((char *)begining,
(int)semicolon);

        if ( !ending )
        goto cq_exit;

        temp = *ending;
        *ending = (UBYTE) 0;
        rtnval = (UWORD) atol((char *)begining);
        *ending = temp;
```

```
cq_exit:
        return(rtnval);
}


/* ubtoa - Convert an unsigned byte to an ascii string
(decimal). */

#asm

        public _ubtoa

_ubtoa:
        movem.l  d0-d1/a0,-(sp)
        and.l    #$ff,d0
        moveq    #2,d1
1$ move.b        #$20,(a0)+
        dbra     d1,1$
        moveq    #2,d1
2$ divu #10,d0
        swap     d0
        or.b     #$30,d0
        move.b   d0,-(a0)
        swap     d0
        ext.l    d0
        dbra     d1,2$
        movem.l  (sp)+,d0-d1/a0
        rts

#endasm

/* uwtoa - Convert an unsigned word to an ascii string
(decimal). */

#asm

        public _uwtoa

_uwtoa:
        movem.l  d0-d1/a0,-(sp)
        and.l    #$ffff,d0
```

```
        moveq    #4,d1
3$ move.b        #$20,(a0)+
        dbra     d1,3$
        moveq    #4,d1
4$ divu #10,d0
        swap     d0
        or.b     #$30,d0
        move.b   d0,-(a0)
        swap     d0
        ext.l    d0
        dbra     d1,4$
        movem.l  (sp)+,d0-d1/a0
        rts

#endasm

/* ultoh - Convert an unsigned long to its ascii string
(hex). */

#asm

        public _ultoh

_ultoh:
        movem.l  d0-d2/a0,-(sp)
        moveq    #7,d2
5$ move.b        #$20,(a0)+
        dbra     d2,5$
        moveq    #7,d2
        moveq    #0,d1
6$ move.b        d0,d1
        lsr.l    #4,d0
        and.b    #$f,d1
        or.b     #$30,d1
        cmp.b    #$39,d1
        ble.s    7$
        addq.b   #7,d1
7$ move.b        d1,-(a0)
        dbra     d2,6$
        movem.l  (sp)+,d0-d2/a0
        rts

#endasm
```

# AMAZING COMPUTING

## AC's TECH

# Back Issue Index

**What have you been missing?** Have you missed information on how to add ports to your Amiga for under $70, how to work around *DeluxePaint*'s lack of HAM support, how to deal with service bureaus, or how to put your Super 8 films on video tape, along with Amiga graphics? Do you know the differences among the big three DTP programs for the Amiga? Does the ARexx interface still puzzle you? Do you know when it's better to you use the CLI? Would you like to know how to go about publishing a newsletter? Do you take full advantage of your RAMdisk? Have you yet to install an IBM mouse to work with your bridgeboard? Do you know there's an alternative to high-cost word processors? Do you still struggle through your directories?

Or if you're a programmer or technical type, do you understand how to add 512K RAM to your 1MB A500 for a cost of only $30? Or how to program the Amiga's GUI in C? Would you like the instructions for building your own variable rapid-fire joystick or a 246-grayscale SCSI interface for your Amiga? Do you use easy routines for performing floppy access without the aid of the operating system? How much do you really understand about ray tracing? **The answers to these questions and others can be found in** *AMAZING COMPUTING* **and** *AC's TECH.*

## How to place your order

We accept Visa and Master Card. Call our toll-free 800 number from anywhere in the U.S. or Canada today!

# 1-800-345-3360

# Programming the Amiga in Assembly Language

## Part 3— Getting Graphic!

*by William P. Nee*

In the last two articles I've discussed libraries, macros, and several of the internal routines. Now it's time to explain one of the Amiga's best features—graphics. Just as in Basic, graphics programs usually start with a SCREEN and WINDOW command. Unfortunately, however, it is not quite that easy. You must give the Amiga a list of your SCREEN and WINDOW specifications (usually called "parameters") and, if this list meets the Amiga criteria, it will create a SCREEN and WINDOW and let you know where they are. Now let's write a simple program, Listing 1, that opens a SCREEN and WINDOW and then draws some graphics.

## SCREENS

We'll start with the SCREEN description first. The items that must be included are:

| PARAMETERS | LENGTH | DESCRIPTION |
|---|---|---|
| left edge | word | left side of screen, usually 0 |
| top edge | word | top of screen, usually 0 |
| width | word | usually 320 or 640 |
| height | word | usually 200 or 400 |
| depth | word | number of bitplanes (1-6) |
| detailpen# | byte | color of gadgets and title bar text |
| blockpen# | byte | block fills - title box, etc. |
| flags | word | combinations of: |
| | | low resolution = $0 |
| | | high resolution = $8000 |
| | | interlaced = $4 |
| | | ham = $800 |
| | | extra half-brite = $80 |
| | | dual play-field = $4000 |
| type screen | word | 1 = Workbench screen; $F = custom |
| screen | | |
| font | long | pointer to other font if any; else 0 |
| title | long | pointer to title if any; else 0 |
| gadget | long | pointer to gadget if any; else 0 |
| custom bitmap | long | pointer to bitmap if any; else 0 |

If you look near the end of Listing 1 you will see a section called "myscreen" where this data is located. The "dc." means that this location in memory has the following values and is "b", "w", or "l" in length. To open our screen, all we need to pass is the location of this data as "myscreen". Two of the data, "depth" and "custom screen", were defined at the beginning of the listing in the "equates:" section. This makes it easier to change values that are scattered throughout the listing since they will always have the value defined at the beginning.

The sub-routine for opening a screen is located in the Intuition library, so we'll open that library first. Knowing the location of the Intuition library, we can now open a screen with the OpenScreen sub-routine. The address of our screen data ("myscreen") is stored in register a1; the library location goes in a6 (generally any library location gets stored in a6), and we JSR to the OpenScreen offset. If the routine can't open a screen, d0 will contain 0 or else it will contain the address of the Amiga screen structure (346 bytes of information - see Table 1).

## WINDOWS

We'll also have to open a WINDOW. The minimum requirements to include are:

| PARAMETER | LENGTH | DESCRIPTION |
|---|---|---|
| left edge | word | left side of window, usually 0 |
| top edge | word | top of window, usually 0 |
| width | word | usually 320 or 640 |
| height | word | usually 200 or 400 |
| detailpen# | byte | same as in screen |
| blockpen# | byte | same as in screen |
| IDCMP flags | long | |
| window flags | long | |
| gadget | long | pointer to gadget if any; else 0 |
| checkmark | long | location if any; else 0 |
| title | long | pointer to title if any; else 0 |
| screen | long | screen location or 0 |
| bit-map | long | pointer to bit-map if any; else 0 |
| min width | word | \ only applies when SIZING- |
| min height | word | \ GADGET has been selected; |
| max width | word | / a 0 will duplicate the settings |
| max height | word | / for that dimension |
| type screen | word | 1 = WorkBench; $F = custom screen |

The window is opened using Intuition's OpenWindow routine. Since the "screen" information is still in d0, we can move this to the screen location (+30) in our window data. When the routine has executed properly, d0 will contain the window structure location (128 bytes of window information—see Table 2).

## FLAGS

But what about those IDCMP and window "flags"? They're merely numbers that tell the computer what type of window you want and how you'll stay in contact with Intuition. They can be in the "equates:" portion of the program or actually combined into one value. These flags are:

| IDCMP FLAGS | VALUE | WINDOW FLAGS |
|---|---|---|
| | 0 | smart_refresh |
| size verify | 1 | want drag gadget |
| refresh_windows | 4 | want depth gadget |
| report mouse buttons | 8 | want closing gadget |
| report mouse moves | $10 | sizing gadget on right |
| LMB down on gadget | $20 | sizing gadget on bottom |
| LMB up on gadget | $40 | simple_refresh |
| requester set | $80 | super_bitmap |
| report menu picked | $100 | always backdrop |
| report close-window picked | $200 | report mouse activity |
| raw key | $400 | gimmezerozero |
| requester verify | $800 | borderless window |
| requester removed | $1000 | activate when open |
| menu verify | $2000 | set by Intuition |
| new preferences | $4000 | set by Intuition |
| report disk inserted | $8000 | set by Intuition |
| report disk removed | $10000 | RMB trap |
| WorkBench message | $20000 | no_care_refresh |
| active window | $40000 | |
| inactive window | $80000 | |
| delta move | $100000 | |
| vanilla key | $200000 | |
| intuiticks | $400000 | |

For window flags you could use:

```
    dc.l BORDERLESS!ACTIVATE (if defined in "equates:")
or dc.l $1800
```

Once we have the window information, one major piece of information can be obtained—the window's RastPort (rp). This area of 100 bytes (see Table 3) is located 50 bytes away from the window address just returned in d0. The RastPort includes all pen colors, line patterns, and area patterns information. Its location is required in most graphics sub-routines so we might as well get it now.

## A GRAPHICS DEMO

Let's review the entire program, Listing 1. In the "equates:" portion JAM1, JAM2, COMPLIMENT, and INVERSE refer to the way text is written on the screen. We'll experiment with this in a future article. The "offsets:" portion includes the offsets for the Exec, Intuition, and Graphics libraries along with a brief description of what must go in the required registers. Finally, I've assigned the variables "across" and "down" to registers d6 and d5 rather than reserve space for them at the end of the program. Any time you can use registers instead of memory locations your program will run more quickly.

The first command line moves the SP address (the current address) to "stack" and saves it there. At the end of the program we'll put that address back in the SP and everything will be as it was. Next, the Intuition and Graphics libraries are opened, then we set up the screen and window, save the RastPort location, and use SetDrMd to set our text style to JAM1.

The graphics demo sets the APen (foreground) color to 3 with the SetAPen routine. Next we'll fill a 60x60 block with this color. When you use WritePixel to PSET a point, registers d0 and d1 need to be clear of everything except the coordinates. Since we're only changing one word, you can make sure the other word is a zero by EXTending the sign through the last 16 bits. Remember that the sign of a positive number is 0. This could also have been done first with MOVEQ #0,d0 since that clears the entire register. ADD will increase the contents of a register by the amount designated; ADDQ (or SUBQ) can be used if this value is

from 1-8. The CMP command will compare a value and a register or two registers, and set the conditional flags according to the result. If the two values are not equal, a BNE (Branch if Not Equal) command will branch; BNE.S means it is a short branch (-128 to +127 bytes away).

Fill_rectangle first changes the pen color to 4 and then uses RectFill to draw and almost instantly fill a box with color. The draw_circle routine first changes the outline pen color to 4. There is no ROM routine for this, so we have to go directly to the RastPort and put a 4 at the OPen location 27 bytes from there. You'll see that the other pen routines work this way if you disassemble their commands. The DrawEllipse routine is then used to draw a circle by making the two radii in d2 and d3 the same. After the APen color has been switched to 3, the Flood routine fills the circle just as in Basic's PAINT command. Mode0 means to fill the circle until the outline color is reached.

The last routine draws a diamond within a diamond and then fills the spaces with different colors. After setting both the OPen and APen color to 2, the Move routine moves to location 250,25. If you didn't use Move to go there, there would be a line from your last drawn location to that point. PolyDraw connects lines from a table of coordinates (PolyTable1 and PolyTable2) at the end of the program. The table location goes in register a0 and the number of pairs in d0. Note that you have to repeat the first pair as the last pair to connect the lines. Then the computer moves to 250,45 and draws the second diamond. After filling the outer space with the APen color, the color is changed to 3 and the inner space is filled.

The "mousepress" portion actually checks just one bit (BTST) in location $BFE001. As long as bit6 in this location is 1, the LMB (Left Mouse Button) has not been pressed and the routine branches back; when you press the LMB, bit6 changes to 0 and the program continues. Close the window with Intuition's CloseWindow routine and close the screen with CloseScreen. Windows, screens, and libraries are normally closed in the reverse order in which they were opened. The original starting location is placed back in the SP register and the RTS (ReTurn from Sub-routine) takes us back to the CLI. Closing a screen before closing a window will usually cause a crash.

At the end of the program six locations were reserved as long words (dc.l) for variables, followed by the two libraries we needed, the screen and window parameters, and the two polygon tables. Copy or assemble this listing to your PROGRAMS disk (as discussed previously in Part I of this series) as GFXDEMO.ASM and GFXDEMO. Run the program from the CLI as GFXDEMO; pressing the LMB will return you to the CLI. Try changing parts of the program, especially the window flags and IDCMP flags to see how they affect the window; you might even want to add a "my_title".

## USING ARRAYS

This article is also a discussion of arrays—that's the Basic command DIM(X,Y) which sets aside a block of memory X bytes wide and Y bytes deep. Since DIM or arrays usually start at 0, the actual size we're reserving is $(X+1)*(Y+1)-1$ bytes. You do this in assembly language with the Exec routine AllocMem. The size, in bytes, of the array goes in register d0 and the type of memory in register d1. Memory types are:

```
Public    ($1)      - either fast or chip memory
Chip ($2) - uses the lower 512K (graphics and sound)
Fast ($4) - above the first 512K
Clear ($10000) - used with the above to clear all values to 0
```

After the AllocMem routine, d0 will contain the address of where the computer has stored this memory for you. Since the next program needs two arrays, these locations get stored in array1 and array2. A returned value of 0 means the computer couldn't reserve enough space for the memory type you wanted. This routine is part of the EXECMACROS.i included on this disk. All allocated memory must be released using the Exec routine FreeMem—also part of EXECMACROS.i.

## A FASTER PSET

Before we review the next program, one more item. In Listing 1, I used WritePixel to PSET a point. Unfortunately, this frequently used routine is also one of the Amiga's slowest. Let's discuss how PSET actually works and then write our own routine—longer but quicker.

Colors (or their PALETTE) are stored in bitplanes on the screen. Each bitplane is the size of the screen and there may be up to six of them. Think of them as individual arrays with each cell in that array containing the value of one bit of the color number. The first bitplane holds the first color number bit, the second bitplane holds the second color number bit, etc. If the color number is 23 (10111), the first bitplane will have a 1, the second a 1, the third a 1, the fourth a 0, and the fifth a 1. Now, first of all, where are the bitplanes? There is a bitmap address in the RastPort, four bytes in (see Table 3). The bitplane addresses themselves begin eight bytes from the bitmap address and are long words (every four bytes). To obtain all these locations use:

```
        BASIC                            ASSEMBLY
RP&=WINDOW(8)                   MOVEA.L  RP,A1
MAP&=PEEKL(RP&+4)               MOVEA.L  4(A1),A1
PLANE1&=PEEKL(MAP&+8)           MOVE.L   8(A1),PLANE1
PLANE2&=PEEKL(MAP&+12)          MOVE.L   12(A1),PLANE2 ...etc.
```

This would continue up to plane5 or plane6 for HAM. Next, where is any point in relation to these bitplanes? In a low resolution screen each bitplane and the screen itself are 320 bits across and 200 bits down. Since eight bits make one byte, there are 40 bytes (0-39) across—so the byte at the start of a row containing the (across,down) location is 40*down. The byte within that row must be across\8 because there are eight bits per byte. Since bits are labeled 0-7, AND the across distance with 7 to get a number within that range. Unfortunately, bits are numbered from right to left so we have to subtract this value from 7 to get the actual bit.

The location we'll work with is the bitplane address plus 40*down plus across\8, and we have to test the computed bit within that byte. Fortunately, there are three commands that will help:

```
BTST - test a color bit to see if it is 0 or 1
BSET - set a bitplane bit to 1
BCLR - clear a bitplane bit to 0
```

This must be repeated for each bitplane. Although the routine is a lot longer than WritePixel, the built-in ROM routine must check so much else (window placement, etc.) that this routine executes much more quickly!

## AN ARRAY DEMO

Now for the program, Listing 2. We'll start off with an array 161x161 (0-160) and put the value 31 in the center square (161*161-1)/2. Then the program starts modifying the array by going to each cell and adding up the sum of its eight neighbors plus itself. The result is divided by 8 and ANDed with 31 to keep it within 0-31. This new value is put into the corresponding cell of array2. When all the cells in array1 have been checked, the value of each cell in array2 is put back in array1 and each cell is PSET on the screen according to its new value. This continues until you press the LMB. The picture starts out very small in the center of the screen and expands into an egg shape.

After defining the "equates:" and "offsets:", the three variables "sum", "across", and "down" are equated to registers d7, d6, and d5. The program then opens the libraries, screen, and window; notice how macros let us do all this using only four lines. After the draw mode is set, the five bitplane addresses are computed. Once we've loaded the address of variable bitplane1 into register a0, we can automatically increase that address with (a0)+. The "+" will increase the address by the MOVE amount such as MOVE.B by 1, MOVE.W by 2, and MOVE.L by 4. The opposite of this would be -(a0) in which case the address would be decreased by 1, 2, or 4. Increases (postincrement) happen after the MOVE command, decreases (predecrement) before the MOVE. Then space is reserved for the two 26,000 byte arrays.

After storing the location of array1 in a4, the center of the array (12960) is stored in d0 and the value 31 stored at that location. The 0(a4,d0.w) means to add together the value 0, the value in a4, and the value in d0. The value of array1 is again put into a4, and array2 into a5. Each value is increased by 162 since we are actually starting at location (1,1). In fact, we'll always stay one cell away from the edges since we want to change/check values one square out in all directions from the cell we're in. If we started right at the top row, we'd be using squares outside the array. This is also why "down" and "across" are only 158 during the first pass.

Whatever cell we're in, the cell -162 bytes away is the cell one row above and one to the left; that cell's value is put in "sum". The cell just above us, or -161 away, and its value are added to "sum"; the cell above and to the right is -160 away and its value is added to "sum". The cells -1, +1, +160, +161, +162 bytes away, and the cell itself all have their values added to "sum". "Sum" is divided by 8 using a shift command and then ANDed with 31 to keep it within the color values.

Shifts are a rapid way to multiply or divide by powers of 2. Each shift of a register to the right will divide its contents by 2 and each shift to the left will multiply it by 2. Shifts only apply to data registers. If you specify a shift number, it must be between 1-7; if you use a data register to hold the number of shifts, it may be between 1-63. The bit that is shifted out of the register usually goes to the carry flag. In general, the three types of shifts are:

```
ASL - left bit goes to carry flag, right bit = 0
ASR - left bit stays the same (sign), right bit to carry flag
LSL - same as ASL
LSR - left bit = 0, right bit to carry flag
ROL - left bit goes to carry flag and to bit0
ROR - right bit goes to carry flag and to bit7
```

Only a rotate command will eventually go full-circle back to its original value.

Once the new value has been computed it's stored in array2 and the location is automatically increased by 1 (MOVE.B SUM,(A5+)); LEA is used to increase the location in array1 by 1. The DBF command (Decrease, Branch if Faults) means to keep decreasing the down value and, as long as it is 0 or greater, branch to L2—a very quick counter.

At the end of the first pass the array locations are again put in a4 and a5. The first contents of each array are compared, and if they're the same, the program branches. If they're not the same, the contents of array2 are moved to "sum" and array1. Note the use of predecrement addressing (-(a5) and -(a4)). The qpset routine sets the points at (across,down) with the color value in sum. To center the display of 160x160, I added 80 to the across location and 20 to the down location. The down location is multiplied by 40 through the use of shifts—down*32 + down*8 = down*40. Since the screen depth is 5, there are five loops within this routine to check each color bit and set/reset the corresponding bit of each bitplane. At the end of qpset each array counter is increased by 1.

When the second pass is finished, the program checks to see if you've pressed the LMB. If not, it will branch back to showit. There are two "showit" locations. The first one always puts #31 in the center square. You can change the program by putting semicolons in front of those four lines and removing the semicolon from the next "showit". There is also a line in the L1 routine where you can remove the semicolon for a major change in the display.

The program ends by first freeing the memory in array1 and array2 with the Exec routine FreeMem. If you don't do this, that memory stays reserved until you power down. Next, the window, screen, and libraries are closed in the reverse order in which they were opened. The data at the end of the program reserves space for variables, defines libraries, and sets window and screen parameters.

Feel free to experiment with this program. Try just adding up the four squares that border the center cell and divide the result

by 4 (LSR.W #2,SUM). Add a value other than 1 to the result or eliminate using the center square. The variations are endless when working with arrays—you can set your own rules in almost any possible combination. Copy or assemble this listing to your PROGRAMS disk as EGG.ASM and as EGG.

## INTUITION MACROS

Now let's write some macros that will make using screens, windows, and graphics a lot easier. Both of these macro files, along with the EXECMACROS.i are included on the magazine disk. Since the Intuition library is used first to open screens and windows, it makes sense to start with it. In Listing 3 I've started with the offsets for the various Intuition routines and their register requirements. The "equates:" portion will help in setting up your screen and window requirements. For a 640x400 screen you could include either $8004 as the mode or type HIRES!INTERLACE (the "!" means to logically OR the values for HIRES and INTERLACE). A borderless window with sizing, drag, depth, and closing gadgets could be described in Flags as $80F or as

```
WINDOWSIZING!WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSING!BORDERLESS.
```

In both cases, the second method takes longer to type but is easier to read and change—it's your choice.

The macros are pretty obvious. OPENWINDOW will insert the screen location into "mywindow"; once it has opened the window the RastPort is stored in RP and the ViewPort is stored in VP. The ViewPort (Table 4) is used mainly when you want to change PEN and PALETTE colors. GETBITPLANES will store six bitplane locations, although the values are only valid up to the DEPTH you've selected. To make it easier, all of the storage locations are within the macro so you don't have to worry about them when writing your program. Copy or save this listing to your ASSEMBLER disk (as discussed in Part I of this series) as INTMACROS.i.

## GRAPHICS MACROS

The major routines, their offsets, and register requirements are listed first. Pen colors are set using FOREGROUND or BACKGROUND. OUTLINE sets both the outline and foreground pens to the same color as required by the Flood routine. Whenever possible, I like to use Basic commands and format as macros; I find them easier to use and the macro requirements easier to remember. That's why I call the macro using the Move routine LOCATE and the one using WritePixel, PSET.

My QPSET is the fast PSET we've already discussed. Both PSET and QPSET allow you to include an Xoffset and Yoffset to center the picture; also both assume that "across" and "down" will hold the X- and Y-coordinates. QPSET requires that DEPTH be equated to a value (1-6), the color to be passed with the macro, and is only for a low resolution picture (320x200 or 320x400). How would you convert this macro to show a high resolution (640 across) picture?

## TABLE 1
## SCREEN STRUCTURE (346 BYTES)

| | | | |
|---|---|---|---|
| 0 | POINTER TO NEXT SCREEN | 35 | WINDOW TOP BORDER |
| 4 | POINTER TO FIRST WINDOW | 36 | WINDOW LEFT BORDER |
| 8 | LEFT EDGE | 37 | WINDOW RIGHT BORDER |
| 10 | TOP EDGE | 38 | WINDOW BOTTOM BORDER |
| 12 | WIDTH | 40 | POINTER TO FONTS |
| 14 | HEIGHT | 44 | VIEWPORT STRUCTURE |
| 16 | MOUSEX | 84 | RASTPORT STRUCTURE |
| 18 | MOUSEY | 184 | BITMAP STRUCTURE |
| 20 | INTUITION FLAGS | 224 | LAYERINFO STRUCTURE |
| 22 | POINTER TO SCREEN TITLE | 326 | POINTER TO FIRST GADGET |
| 26 | DEFAULT TITLE | 330 | DETAIL PEN (BORDER) |
| 30 | BAR HEIGHT | 331 | BLOCK PEN (MENU, DRAGBAR) |
| 31 | BAR VERTICAL BORDER | 332 | SAVECOLOR0 |
| 32 | BAR HORIZONTAL BORDER | 334 | POINTER TO BARLAYER |
| 33 | MENU VERTICAL BORDER | 338 | POINTER TO EXTERNAL DATA |
| 34 | MENU HEIGHT BORDER | 342 | POINTER TO USER DATA |

PALETTE uses the LoadRGB4 routine to change the actual colors of the individual pens. We'll talk about this macro in the next program. PCLS will clear the screen to color0 or a color passed with the macro. LINE draws a line connecting X1,Y1 and X2,Y2; the color is an optional value. BOX will draw a rectangle between X1,Y1 and X2,Y2; again, color is an optional value. BOXF will draw a box and rapidly fill it using the RectFill routine.

CIRCLE and ELLIPSE both use the DrawEllipse routine. If the two radii are the same, the result will be a circle; the color is optional in both macros. PEN will change the color of an individual pen#. This change will also apply to anywhere that color is already on the screen—useful for fade-in/out or animation. PAINT uses the Flood routine to fill an area with the foreground color. The macro assumes that you want mode0 (fill until you reach the outline color) unless you pass a "1" in the macro. Finally, POLYGON will connect the coordinates listed in a table; the color is an optional value. Copy or save this listing on your ASSEMBLER disk as GFXMACROS.i.

Try re-doing Listing 1 using as many of these macros as possible to replace the routines in the listing. I would start with the graphics macros first and then substitute the Intuition ones. When you feel comfortable with these macros, feel free to rename them, modify them, or make whatever changes you want. Just remember, the more complicated you make them, the harder they are to remember a week later.

## PASSING VALUES

Next I'll show you how to pass values from the CLI to your program, make your own semi-random function, and change the

color palette. The demonstration program is a version of the "Demon" article in *Scientific American*, where cells devour other cells of a lower value. As in the previous program, I'll use a 160X160 array centered in a low resolution screen.

In the "Demon" program the four neighbors around a cell are checked to see if any of them has a value one greater than the center cell. If so, the center cell is "eaten" (replaced) by the new value and the next cell is checked. Cell values wrap around so that the maximum value plus 1 equals zero. The default maximum value is 15, but you may pick any number up to 31. However, values above 20 tend to fade out quickly since the size of the array isn't that large. Initially, I'd experiment with values from 10-20. As cells devour each other, the pattern of the random array becomes more regular and may gradually change to spirals that take over the entire picture.

Since the "Demon" program needs to know the maximum cell value you want to use, we'll pass that value (1-31) from the CLI along with the program name. Register a0 is always a pointer to the address where anything after a CLI command is stored (as an ASCII character string) and register d0 contains the number of characters.

Follow the "get_CLI_value" routine in Listing 5 and you can see that the first step is to save both the string address and number of characters in separate registers. If the number of characters minus 1 is zero, then all you typed in was the program name, and the routine automatically passes a default value of 15 to the variable "level". When there is another character though, it must be a number, so subtract #\$30 from its ASCII value to get the actual number.

Subtract 3 from the value in d5. If the result is 0, there are no more characters and we have only a value from 1-9 to put in "level". But if there is still another character, then the one we already have is the ten's portion, so multiply it by 10. MULU means to MULtiply Unsigned values (MULS will MULtiply Signed values). Now add the next character to the result and again subtract #\$30 to normalize the result. Store this value in the variable "level".

## A RANDOM ROUTINE

There is no specific random routine in assembly language so you have to adapt one for yourself. Mine uses one of the Complex Interface Adaptor (CIA) registers at \$BFE801 that normally counts events. Follow the "random" routine in Listing 5 and you can see that the contents of \$BFE801 are stored in d4; then various changes are made to this value. Since we want a random number between 0 and the value in your CLI command (or the default value of 15), the value in d4 must always be compared with the value in "level". The program keeps looping until an acceptable number is found. The NOP command means No OPeration and is just a delay to keep the events counter "ticking". If you have your own favorite random routine, feel free to substitute it for this one. Be sure to keep the result between 0 and "level", then store the result in d4.

---

## TABLE 2
## WINDOW STRUCTURE (128 BYTES)

| | |
|---|---|
| 0 NEXT WINDOW POINTER | 62 GADGET POINTER |
| 4 LEFT EDGE | 66 PARENT (OPEN/CLOSE) |
| 6 TOP EDGE | 70 DESCENDANT |
| 8 WIDTH | 74 POINTER TO SPRITE DATA |
| 10 HEIGHT | 78 POINTER SPRITE HEIGHT |
| 12 MOUSEX | 79 POINTER SPRITE WIDTH |
| 14 MOUSEY | 80 POINTER XOFFSET |
| 16 MINIMUM WIDTH | 81 POINTER YOFFSET |
| 18 MINIMUM HEIGHT | 82 IDCMP FLAGS |
| 20 MAXIMUM WIDTH | 86 POINTER TO USERPORT |
| 22 MAXIMUM HEIGHT | 90 POINTER TO MSGPORT |
| 24 FLAGS | 94 POINTER TO IMESSAGE |
| 28 MENU STRUCTURE | 98 DETAIL PEN# |
| 32 POINTER TO TITLE | 99 BLOCK PEN# |
| 36 FIRST REQUESTER POINTER | 100 CHECKMARK POINTER |
| 40 DOUBLE CLICK REQUESTER | 104 SCREEN TITLE POINTER |
| 44 COUNT OF REQUESTERS | 108 GIMMEZEROZERO MOUSEX |
| 46 SCREEN POINTER | 110 GIMMEZEROZERO MOUSEY |
| 50 RASTPORT POINTER | 112 GIMMEZEROZERO WIDTH |
| 54 LEFT BORDER WIDTH | 114 GIMMEZEROZERO HEIGHT |
| 55 TOP BORDER WIDTH | 116 EXTERNAL DATA POINTER |
| 56 RIGHT BORDER WIDTH | 120 USER DATA POINTER |
| 57 BOTTOM BORDER WIDTH | 124 DUPLICATE LAYER POINTER |
| 58 BORDER RASTPORT | |

I wanted a different palette for the first 16 colors (0-15), so I used the PALETTE macro to create a new one from the color table at the end of the program. The value of each word in the table represents the amount (0-F) of red, green, and blue, respectively, in that pen#. You can change these to any color you want. Notice that I have the first color as 0,0,0 so Pen0, the background, will be black.

To initiate PALETTE you need the ViewPort structure location. Unlike obtaining the RastPort location there is a routine to get the ViewPort—Intuition's ViewPortAddress. All you need to do is store the window structure address in register a0 and call the routine; register d0 will contain the ViewPort address. To change the palette put the ViewPort address in a0, the location of your color table in a1, and the number of pens you're changing in d0. Pen changes always start with Pen0.

## THE DEMON PROGRAM

Now let's go through Listing 5. In addition to the routines just described, there are two new routines from the Exec library—Forbid and Permit. Forbid will stop all other activity, mouse movement, etc.; this will enable the program to run a little more quickly. The opposite, Permit, frees the computer and allows multi-tasking.

I've added the register equate "sum" since it is used so often during the program. The size of the arrays we'll use is also defined as 26000. There are now FOREGROUND and PSET macros but, with QPSET available, we won't use them too often. They are used, however, after the random routine gets a value; that value is used to set the APen and color the location. The Exec macro ARRAY uses the previously defined size to reserve memory and

---

### TABLE 3
### RASTPORT STRUCTURE (FIRST 40 BYTES)

```
0  LAYER POINTER
4  BITMAP POINTER
8  AREAPTRN - pointer to areafill pattern
12 TMPRAS - temporary rastport or 0
16 AREAINFO POINTER
20 GELSINFO POINTER
24 MASK - usually 255
25 FOREGROUND - SetAPen
26 BACKGROUND - SetBPen
27 OUTLINE PEN - use SetOPen macro
28 DRAW MODE - jam1=0, jam2=1, compliment=2,
   inversevid=4
29 AREA POINTER SIZE - a power of 2
30 LINE PATTERN COUNT
31 DUMMY
32 FLAGS
34 LINE PATTERN - solid line = -1
36 CURRENT PEN X
38 CURRENT PEN Y
```

---

### TABLE 4
### VIEWPORT STRUCTURE (40 BYTES)

```
0  NEXT VIEWPORT POINTER          26 D HEIGHT
4  COLORMAP POINTER               28 DX OFFSET
8  DSPINS - used by MakeView      30 DY OFFSET
12 SPRITEINS - used by Sprites    32 MODES
16 CLRINS - used by Sprites       34 SPRITE PRIORITIES
20 UCOPINS - user Copper List     35 RESERVED
24 D WIDTH                        36 RASINFO POINTER


RASINFO STRUCTURE (12 BYTES)
0  VIEWPORT POINTER
4  LOF COPPER LIST - interlaced and non-interlaced
8  SHF COPPER LIST - only during interlaced
12 DY OFFSET - +/- adjustment to standard offsets
14 DX OFFSET - +/- adjustment to standard offsets
16 MODES


VIEW STRUCTURE (18 BYTES)
0  NEXT RASINFO - for dual playfield
4  BITMAP POINTER
8  RX OFFSET - scroll offset
10 RY OFFSET - scroll offset
```

either stores the memory address in the location passed or branches if unsuccessful to the location given.

After saving the Stack Pointer, the program computes your value passed from the CLI or defaults to 15. The best values to use are within the range of 10-20; below that the pattern looks mainly like small worms, and values above will probably die out due to the small size of the array. The value you select represents the number of colors used. Whenever a center cell's adjacent neighbor has a value one higher than itself, that neighbor can "eat" the center cell by replacing it with its new value. To "wrap" the colors around, 1 is greater then 0 and 0 is greater than the highest value.

Next, the program opens the libraries and sets up the screen and window. Knowing the window structure, the program can obtain the RastPort (rp) and ViewPort (vp). After getting the vp, the program reads the new pen colors and uses PALETTE to change the colors. Then the bitplanes are located and memory is reserved for the two 160X160 arrays. The random portion computes a number from 0 to level, PSETs the (across,down) location to the color value, and stores this value in both array1 and array2. The postincrement mode is used to increase the array locations. After this, the Forbid routine is called.

The demonstration part of this program starts at (1,1) in each array and saves the value there as "sum". Then it adds 1 to this value, wrapping back to 0 if necessary. This is now the value to which the four neighbors touching the center cell will be compared. The cell just above is -161 bytes away (-161(a4)) and it's value is compared to "sum". If it is not the same, the program goes to the cell just to the left (-1(a4)). If, however, they are the same, "sum" is transferred to array2 as the new value. After its neighbors have been checked, each array location is increased by 1 (LEA 1(a4),a4); notice that we can't combine the offset and postincrement modes. At the end of each row both arrays are increased by 2 to compensate for the bytes not checked at the end of one row and the beginning of the next row.

After examining the cells in array1, the contents of array2 are compared to array1; if they're the same the program goes on to the next cell. If they're not the same, however, the value from array2 is transferred to array1 and the corresponding point is set using QPSET with offsets to center the picture. When all of array2 has been checked, the program looks to see if you've pressed the LMB. If not, it repeats the entire process.

To end the program a call is first made to Permit, then the memory in both arrays is cleared with FREE, the window and screen closed, the libraries closed, and the Stack Pointer returned. In addition to the usual variables and window/screen parameters, the program has a color table for the new pen colors. Copy or assemble this program as DEMON.ASM and DEMON. Run this program from the CLI as DEMON, adding the color value you want to use, if any (e.g., DEMON 16).

Besides the changes I've mentioned, you might want to modify the "get_CLI_value" routine by adding another check for incorrect values or mistakes. Try increasing the screen size to interlaced ($4) and then maybe increase the array size to 160X320. Be sure to increase SIZE and the down dimensions within the program; with a larger array you could use higher color values. If you're really ambitious, add a flag that will stop the program and let you know if a pass did not change any cell values and the program is actually over.

## NEXT TIME

In the next article I'll combine assembly language programs with Basic, and show you how to use LHARC in your startup-sequence. We'll review the rules of LIFE, visualize some WALL-PAPER, and experiment with double-buffering.

## Listing 1

```
LISTING 1
;GFXDEMO.asm
equates:
depth      = $5
jam1 = $0
jam2 = $1
compliment        = $2
inverse  = $4
borderless       = $800
ww.screen= $2e
rport    = $32
rp.open = $1b
nw.screen= $1e
customscreen      = $f
activate = $1000

offsets:
;exec
openlibrary       = -552   ;a1=libraryname,d0=version
closelibrary      = -414   ;a1=librarybase
;intuition
openscreen        = -198   ;a0=myscreen->
closescreen       = -66    ;a0=screen
openwindow        = -204   ;a0=mywindow->
closewindow       = -72    ;a0=window
;graphics
drawellipse       = -180   ;a1=rp,d0=x,d1=y,d2=r1,d3=r2
flood    = -330   ;a1=rp,d0=x,d1=y,d2=mode
                           ; (0=outline,1=color)
move = -240       ;a1=rp,d0=x,d1=y
polydraw = -336   ;a0=data->,a1=rp,d0=count
rectfill = -306   ;a1=rp,d0=x1,d1=y1,d2=x2,d3=y3
                           ; (x1<x2,y1<y2)
setapen   = -342  ;a1=rp,d0=color (foreground)
setbpen   = -348  ;a1=rp,d0=color (background)
setdrmd   = -354  ;a1=rp,d0=mode
                           ; (JAM1=0,JAM2=1,COMP=2,INV=4)
setrast  = -234   ;a1=rp,do=color (quick CLS)
writepixel        = -324  ;a1=rp,d0=x,d1=y

across    equr    d6
down equr d5

start:
    move.l        sp,stack ;save the starting location

open_libs:                 ;open required libraries
    lea int,a1    ;library address to a1
    moveq         #0,d0    ;any version number
    movea.l       4,a6     ;exec library address
    jsr openlibrary(a6)
    move.l        d0,intbase         ;save it's location
    beq done      ;couldn't find library

    lea gfx,a1    ;open graphics library
    moveq         #0,d0
    movea.l       4,a6
    jsr openlibrary(a6)
    move.l        d0,gfxbase
    beq close_int

set_up:
make_screen:
    lea myscreen,a0        ;my screen parameters
    movea.l       intbase,a6
    jsr openscreen(a6)
    move.l        d0,screen ;save screen address
    beq close_libs         ;stop if couldn't open a screen
```

```
make_window:
    lea    mywindow,a0        ;my window parameters
    move.l    d0,nw.screen(a0)
    movea.l   intbase,a6
    jsr openwindow(a6)
    move.l    d0,window ;save window address
    beq close_screen

    movea.l   d0,a0
    movea.l   rport(a0),a1
    move.l    al,rp    ;window rastport address

    movea.l   rp,a1
    moveq     #jam1,d0
    movea.l   gfxbase,a6
    jsr setdrmd(a6)        ;graphics drawing mode

;program - graphics demo
showit:
    moveq     #3,d0    ;pen color
    movea.l   rp,a1    ;rastport
    movea.l   gfxbase,a6            ;graphics library
    jsr setapen(a6)        ;foreground color

    moveq     #0,down
loop1:
    moveq     #0,across
loop2:
    movea.l   rp,a1
    move.w    across,d0
    ext.l     d0               ;extend sign
    move.l    down,d1
    ext.l     d1
    movea.l   gfxbase,a6
    jsr writepixel(a6) ;pset
    addq.w    #1,across
    cmp.w     #61,across       ;end of a row?
    bne.s     loop2
    addq.w    #1,down
    cmp.w     #61,down  ;end of display?
    bne.s     loop1

;fill_rectangle
    moveq     #4,d0    ;new color
    movea.l   rp,a1
    movea.l   gfxbase,a6
    jsr setapen(a6)

    movea.l   rp,a1
    move.w    #100,d0
    move.w    #0,d1
    move.w    #200,d2
    move.w    #100,d3
    movea.l   gfxbase,a6
    jsr rectfill(a6)

draw_circle:
    movea.l   rp,a1
    move.w    #4,d0
    move.b    d0,rp.open(a1) ;outline pen
    movea.l   rp,a1
    move.w    #160,d0
    move.w    #170,d1
    move.w    #20,d2    ;radius 1
    move.w    #20,d3    ;radius 2
    movea.l   gfxbase,a6
    jsr drawellipse(a6) ;draw a circle
flood_circle:
    movea.l   rp,a1
    moveq     #3,d0
    movea.l   gfxbase,a6
    jsr setapen(a6)

    movea.l   rp,a1
    move.w    #160,d0
    move.w    #170,d1
    moveq     #0,d2    ;mode
    movea.l   gfxbase,a6
    jsr flood(a6)

draw_polygon
    movea.l   rp,a1
    move.w    #2,d0
    move.b    d0,rp.open(a1)

    movea.l   rp,a1
    move.w    #2,d0
    movea.l   gfxbase,a6
    jsr setapen(a6)

    movea.l   rp,a1
    move.w    #250,d0
    move.w    #25,d1
    movea.l   gfxbase,a6
    jsr move(a6)

    movea.l   rp,a1
    lea    polytable1,a0
    moveq     #5,d0
    movea.l   gfxbase,a6
    jsr polydraw(a6)

    movea.l   rp,a1
    move.w    #250,d0
    move.w    #45,d1
    movea.l   gfxbase,a6
    jsr move(a6)

    movea.l   rp,a1
    lea    polytable2,a0
    moveq     #5,d0
    movea.l   gfxbase,a6
    jsr polydraw(a6)

    movea.l   rp,a1
    move.w    #250,d0
    move.w    #30,d1
    moveq     #0,d2
    movea.l   gfxbase,a6
    jsr flood(a6)

    movea.l   rp,a1
    moveq     #3,d0
    movea.l   gfxbase,a6
    jsr setapen(a6)

    movea.l   rp,a1
    move.w    #250,d0
    move.w    #50,d1
    moveq     #0,d2
    movea.l   gfxbase,a6
    jsr flood(a6)

mousepress
    btst #6,$bfe001
    bne.s         mousepress

close_window:
    movea.l       window,a0 ;close the window
    movea.l       intbase,a6
    jsr closewindow(a6)
close_screen:
    movea.l       screen,a0 ;close the screen
    movea.l       intbase,a6
    jsr closescreen(a6)
close_libs:              ;close the libraries
    movea.l       gfxbase,a1
    movea.l       4,a6
    jsr closelibrary(a6)
close_int
    movea.l       intbase,a1
    movea.l       4,a6
    jsr closelibrary(a6)
done:            ;return stack pointer
    move.l       stack,sp
    rts          ;return to CLI

    even
    even
stack    dc.l 0
gfxbase dc.l 0
```

```
intbase dc.l 0
screen   dc.l 0
window   dc.l 0
rp  dc.l 0

gfx dc.b 'graphics.library',0
    even
int dc.b 'intuition.library',0
    even

myscreen
    dc.w 0,0,320,200,depth ;depth is 5 for 5 bit planes
    dc.b 0,1              ;pens
    dc.w 0               ;screen flags=low res
    dc.w customscreen    ;not WB
    dc.l 0,0,0,0  ;no font,title,gadget or bitmap
    even
mywindow
    dc.w 0,0,320,200     ;size
    dc.b 0,1             ;pencolors
    dc.l $8             ;IDCMP flags=mousebuttons
    dc.l $1800         ;windowflags=borderless!activate
    dc.l 0,0         ;no gadgets or checkmark
    dc.l 0          ;no title
    dc.l 0,0       ;no screen or bitmap
    dc.w 0,0,0,0  ;min/max same as size above
    dc.w customscreen   ;not WB
    even

polytable1
    dc.w 250,25
    dc.w 300,100
    dc.w 250,175
    dc.w 200,100
    dc.w 250,25
    even
polytable2
    dc.w 250,45
    dc.w 280,100
    dc.w 250,155
    dc.w 220,100
    dc.w 250,45
    even
    end
```

# Listing 2

```
LISTING 2
;EGG.asm
    include     execmacros.i
equates:
depth   = $5    ;5 bitplanes
jam2 = $1
mousebuttons    = $8
borderless      = $800
ww.screen= $2e
ww.rport = $32
ww.userport     = $56
backdrop = $100
nw.screen= $1e
customscreen    = $f
activate = $1000
rmbtrap  = $10000

offsets:
;intuition
openscreen      = -198
closescreen     = -66
openwindow      = -204
closewindow     = -72
;graphics
setdrmd  = -354

sum  equr d7
across   equr    d6
down equr d5
```

```
size =   26000

macros:
intlib macro            ;(routine)
    movea.l     intbase,a6
    jsr  \1(a6)
    endm

openscreen macro  ;(parameters,screen,beq if 0)
    lea  \1,a0
    intlib      openscreen
    move.l      d0,\2
    beq  \3
    endm

openwindow macro  ;(parameters,window,beq if 0)
    lea  \1,a0
    move.l      screen,nw.screen(a0)
    intlib      openwindow
    move.l      d0,\2
    beq  \3
    endm

gfxlib   macro          ;(routine)
    movea.l     gfxbase,a6
    jsr  \1(a6)
    endm

start:
    move.l      sp,stack

open_libs:              ;open required libraries
    openlib     int,done
    openlib     gfx,close_int

set_up:
make_screen:
    openscreen  myscreen,screen,close_libs

make_window:
    openwindow  mywindow,window,close_screen

    movea.l     d0,a0
    movea.l     ww.rport(a0),a1
    move.l      a1,rp   ;window rastport address

    movea.l     rp,a1
    move.l      #jam2,d0
    gfxlib      setdrmd  ;graphics drawing mode

get_bit_planes:
    movea.l     rp,a1    ;rastport location
    movea.l     4(a1),a1 ;bitmap location
    lea.l       bpl,a0   ;address of bitplane array
    move.l      8(a1),(a0)+        ;address of bitplane 1
    move.l      12(a1),(a0)+       ;address of bitplane 2
    move.l      16(a1),(a0)+       ;address of bitplane 3
    move.l      20(a1),(a0)+       ;address of bitplane 4
    move.l      24(a1),(a0)        ;address of bitplane 5

array:
    move.l      #size,d0 ;size of each array
    move.l      #$10004,d1         ;fast memory!clear
    syslib      allocmem ;find a location
    tst.l       d0
    beq close_window        ;couldn't find a location
    movea.l     d0,a4    ;save the address
    move.l      d0,array1;save it here,too

    move.l      #size,d0
    move.l      #$10004,d1
    syslib      allocmem
    tst.l       d0
    beq close_out
    movea.l     d0,a5
    move.l      d0,array2
    moveq       #0,sum   ;clear
    moveq       #0,across; out
    moveq       #0,down  ;       everything we'll use
    lea.l       bpl,a0
```

```
;program - egg
showit:
    movea.l         array1,a4 ;address of array 1
    move.w          #12960,d0 ;center of array 1
    move.b          #31,0(a4,d0.w) ;store #31 in the center
;showit:                 ;(optional)
    movea.l         array1,a4 ;array 1
    lea 162(a4),a4          ;over 1, down 1
    movea.l         array2,a5 ;array 2
    lea 162(a5),a5          ;over 1, down 1
    move.w          #158,down ;compute 158 rows
12
    move.w          #158,across       ;compute 158 values per row
11
    move.b          -162(a4),sum      ;get sum of 3x3 block
    add.b           -161(a4),sum
    add.b           -160(a4),sum
    add.b           -1(a4),sum
    add.b           0(a4),sum
    add.b           1(a4),sum
    add.b           160(a4),sum
    add.b           161(a4),sum
    add.b           162(a4),sum
    lsr.w           #3,sum   ;divide sum by 8
;   addq #1,sum     ;(optional)
    andi.b          #31,sum  ;keep sum below 32
    move.b          sum,(a5)+ ;store the sum in array 2

    lea 1(a4),a4 ;next value
    dbf across,11 ;branch if row not finished
    lea 2(a4),a4 ;jump two spaces
    lea 2(a5),a5
    dbf down,12 ;branch if not finished

    movea.l         array1,a4 ;array 1
    movea.l         array2,a5 ;array 2
    moveq           #0,down  ;start at the beginning
14
    moveq           #0,across
13
    cmpm.b          (a5)+,(a4)+      ;compare array 1 and 2
    beq nextone     ;skip this if the same
    move.b          -(a5),sum ;transfer the values
    move.b          sum,-(a4)
qpset:
    move.w          down,d0
    add.w           #20,d0   ;center the display down
    move.w          d0,d1
    asl.l           #5,d0    ;down x 32
    asl.l           #3,d1    ;down x  8
    add.w           d1,d0    ;equals down x 40
    movea.w         d0,a2    ;save this
    move.w          across,d0
    addi.w          #80,d0   ;center the display across
    ext.l           d0
    asr.l           #3,d0    ;divide by 8 (bits/byte)
    adda.w          d0,a2    ;actual screen position
    move.w          across,d0
    andi.w          #7,d0    ;only have bits 0-7
    move.w          #7,d2
    sub.w           d0,d2    ;actual bit
    ext.l           d2
    move.w          #depth-1,d3       ;# of bitplanes-1
qploop
    move.w          d3,d0
    ext.l           d0
    asl.l           #2,d0    ;times 4
    move.w          a2,d1    ;actual screen position
    ext.l           d1
    movea.l         0(a0,d0.1),a1    ;current bitplane address
    adda.l          d1,a1    ;actual bitplane location
    btst d3,sum     ;bit value of the color
    bne setit       ;branch if one
    bclr d2,(a1)    ;clear bit if zero
    bra.s           qploop1
setit:
    bset d2,(a1)    ;set bit if one
qploop1:
    dbf d3,qploop ;color all done?
    lea 1(a4),a4 ;move one space over
    lea 1(a5),a5
```

```
nextone:
    addq.w          #1,across
    cmp.w           #161,across          ;end of a row?
    bne 13
    addq.w          #1,down
    cmp.w           #161,down ;end of display?
    bne 14

    lmb showit      ;LMB pressed?
    bne showit      ;loop all the way back

free:
    movea.l         array2,al ;address of array 2
    move.l          #size,d0 ;it's size
    syslib          freemem  ;free the memory
close_out:
    movea.l         array1,al ;ditto for array 1
    move.l          #size,d0
    syslib          freemem

close_window:
    movea.l         window,a0 ;close the window
    intlib          closewindow
close_screen:
    movea.l         screen,a0 ;close the screen
    intlib          closescreen
close_libs:                      ;close the libraries
    closelib gfx
close_int:
    closelib int
done:
    move.l          stack,sp
    rts             ;return to CLI

    evenpc
stack    dc.l       0
gfxbase  dc.l       0
intbase  dc.l       0
screen   dc.l       0
window   dc.l       0
rp dc.l 0
array1   dc.l       0
array2   dc.l       0
bpl ds.l 6

gfx dc.b 'graphics.library',0
    evenpc
int dc.b 'intuition.library',0
    evenpc
myscreen:
    dc.w 0,0,320,200,depth
    dc.b 0,1
    dc.w 0
    dc.w customscreen
    dc.l 0,0,0,0
    evenpc
mywindow:
    dc.w 0,0,320,200
    dc.b 0,1
    dc.l mousebuttons
    dc.l borderless!activate!rmbtrap
    dc.l 0,0,0,0,0
    dc.w 0,0,0,0
    dc.w customscreen
    end
```

☑

**The remainder of the listings for Programming the Amiga in Assembly Language Part III can be found on the AC's TECH disk.**

# BackUp

*by Werther Pirani*

If you do some programming and you need to speed up its development a little, the best thing to do is to use a RAM-DISK to keep your sources, your executables and, if you can afford some extra RAM, even the include files. A recoverable RAM-DISK surely might be your little assurance against a visit from the dreaded GURU. If there was a power failure, you could lose all of your files in RAM. The best thing to do is to make a back-up at regular intervals and copy the files from the RAM-DISK to a floppy disk. Even if you're lazy, this is not too demanding but when you're working on a larger program whose source is broken into smaller files, things can become pretty boring. Even worse, you can't always remember how many files you've modified since the last back-up.

A script file like that in listing 1 might be useful. Although it does back-up only the most recent files, it copies ALL the most recent files, including those that weren't even modified since the previous back-up, so a different solution has to be found. Well, the perfect solution is not too far away and it comes in the shape of the file archive-bit!

Typically, the archive-bit should be set each time a file is backed-up and should be cleared each time a file has been modified. Unfortunately, as of this writing, the file archive-bit is something that AmigaDOS doesn't deal automatically with. That is, while you are in the CLI you can set it directly by typing:

```
Protect  filename  +a
```

or you can do the reverse by typing:

```
Protect  filename  -a
```

but nothing more: even though the Copy and the List commands are quite flexible, especially the latter in conjunction with the LFORMAT option, you can't get them to take into account the status of the file archive-bit. That's why I've come up with this little CLI utility.

## How to Use It

BackUp is essentially a copy program that is directory oriented, rather than file oriented. In fact, it's intended to work with the files within a given directory or within nested directories. You can use it only from the CLI by typing:

```
Backup  SourceDir  DestDir  [ALL]  [QUIET]
```

Of course, SourceDir and DestDir are mandatory and must both exist before the command is executed while the ALL and QUIET switches are optional and have the same functionality as of those from the Copy command. Let's see a few examples:

```
Backup  ""  df1:MyDir
```

Each file within the current directory with the archive bit still clear is copied to df1:MyDir and then its archive bit is set. Information about the operation in progress is printed to the CLI, but nested directories are ignored.

```
Backup  ""  df1:MyDir  ALL
```

Same as above but takes into account nested directories. If such directories don't exist within the destination directory, first they are created and then the files with the archive bit not set are copied to them. Information about the operation in progress is still printed to the CLI.

```
Backup  ""  df1:MyDir  ALL  QUIET
```

Same as above but no information about the operation in progress is printed to the CLI, except for error messages. However, if you want to suppress them, you can use redirection:

```
Backup  >NIL:  ""  df1:MyDir  ALL  QUIET
```

## How It Works

Listing 2 is the source for BackUp and it's fully documented. Rather than discuss it at some length, I'd like to point out a few things about AmigaDOS, but don't expect me to supplant any documentation currently available.

As to begin with, a few words about the lock, a mechanism provided by AmigaDOS for moving around in its file system. When you're locking a file or a directory, you're asking AmigaDOS to refer all of your requests to that particular file or directory. A call to the Lock() function takes this form:

```
lock = Lock(path_string, access_mode);
```

where lock is a pointer to a FileLock structure returned by Lock(), path_string specifies the file or directory you want to lock, for example "df1:MainDir/OtherDir/FinalDir" and access_mode is either ACCESS_READ also called SHARED_LOCK because any other program can access it, or ACCESS_WRITE also called EXCLUSIVE_LOCK because any other program is effectively locked out from the access to it. However, whatever the access_mode you choose, never forget to unlock what you've locked or the AmigaDOS will soon become pretty messed up and you'll have to reboot to bring things back to normality. A call to the UnLock() function takes this form:

```
UnLock(lock);
```

where lock is a pointer returned by the previous call to Lock(). That is, Lock() and UnLock() must always be paired.

If you've locked a directory, then you can move to that directory using the CurrentDir() function whose effect is the same of the CD command:

```
oldlock = CurrentDir(lock);
```

once again, lock is a pointer returned by the Lock() function and oldlock is another pointer to let you come back to wherever you were in the first place:

```
dummylock = CurrentDir(oldlock);
```

Once you've locked a directory and moved to it, you can get information about the Directory Type, FileName, Size, Date, Comment, Protection bits and so on by reading the contents of a FileInfoBlock structure which is listed in the include file libraries/dos.h, but first you must allocate enough memory for it:

```
fib = (struct FileInfoBlock *)
   AllocMem(sizeof(struct FileInfoBlock), MEMF_CLEAR);
```

if fib is not NULL then you can call the Examine() function to fill the FileInfoBlock structure with the information about that directory:

```
success = Examine(lock, fib);
```

where lock is the pointer to a FileLock structure returned by Lock() and fib is the pointer to a FileInfoBlock structure allocated by AllocMem(). If success is not zero then you can examine the contents of that FileInfoBlock and finally access the information you need. Furthermore, if you want to get information about the files within the same directory you can use the ExNext() function:

```
success = ExNext(lock, fib);
```

The parameters are the same as those for the Examine() function but this time a value of zero for success means "no more entries in this directory" rather than "an error has occurred." By the way, while I'm on the subject let me say that if you are a serious programmer, or just an investigative one, you can discover why an AmigaDOS function has failed calling the IoErr() function:

```
error = IoErr();
```

The include file libraries/dos.h contains a complete list of the errors that you can encounter.

AmigaDOS can also perform operations usually accessed from the CLI like rename a file:
```
success = Rename("oldname", "newname");
```

delete a file:
```
success = Delete("filename");
```

protect a file:
```
success = SetProtection("filename", maskvalue);
```

create a directory:
```
lock = CreaterDir("dirname");
```

even establish a filenote:
```
success = SetComment("filename", "comment");
```

However, the most powerful is certainly the Execute() function that works exactly like its CLI counterpart:

```
success = Execute("command", input, output);
```

where command is a string just like the one you'd type from the CLI, i.e. "Dir df1: opt a", while input and output specify how to redirect the standard input and standard output if there is no redirection in the command string. A value of zero for both parameters tells AmigaDOS to use the same standard input and output of the process that calls the function, usually the CLI. A word of warning here: if your program is intended to run only from the CLI, it's okay to use zero for both input and output. If your program could be started also from the Workbench and uses the Execute() function to run any other program in the background, you must provide at least a way to redirect the standard output:

```
output = Open("RAM:dummyfile, MODE_NEWFILE);
success = Execute("Run MyProgram", 0, output);
... your code here, then when it's time to exit...
Close(output);
```

Nonetheless, don't forget that the Execute() function has two main restrictions: the Run command must be present in your C directory and the command that you want to execute must be either in the current or in the C directory.

If you take a closer look at listing 2 you can see that each time we find a file with the archive bit not set, first we build a string like "Copy filename TO pathname CLONE" and then we call the Execute() function to copy it from the source to the destination directory. Of course we could use a series of Read() and Write() to copy it by ourselves but then the information about the date and protection bits would get lost. Unfortunately, since the Copy command must be loaded for each file, this might result in a somewhat slower operation, especially if there are a lot of files to back-up. Nevertheless, if you're running AmigaDOS 1.3 or higher, at the beginning of your working session you can make the Copy command resident by typing:

```
Resident SYS:C/Copy add
```

and everything will be just fine.

An in-depth discussion about the inner workings of AmigaDOS was far beyond the scope of this article and in fact I've barely scratched the surface. If you are a novice programmer and you're willing to learn more, a book that's definitely a must buy is SYBEX' "Programmer's Guide To The Amiga" by Robert A. Peck, a too soon passed away fellow who did a great job for the Amiga community. Needless to say, his work was and still remains valuable today.

## LISTING 1

```
.key from/a,to/a; don't remove from here!!!

;This is a simple script to back-up the most recent files,
;hence the TODAY option in the LIST command, from a source
;directory to a destination directory. Put this file in
;your S: directory and name it as you prefer. Usage is:
;
;     execute scriptname sourcedir destdir
;
;If you're using a shell like the one supplied with
;Workbench 1.3 instead of the old CLI, you can set its
;script-bit to tell AmigaDOS that this one is a script:
;
;     protect scriptname +s
```

```
;
;And when you need a back-up just type:
;
;    scriptname sourcedir destdir


;Please change all references from RAM: to VD0:, RAD: or
;whatever the RAM-DISK you're using...

IF NOT exists RAM:T
   MAKEDIR RAM:T
ENDIF

;Don't break the following line!

LIST > RAM:T/Temp <from> LFORMAT="Copy %s%s TO <to> CLONE"
FILES SINCE TODAY

Echo "Please wait, macking a back-up..."
Execute RAM:T/Temp
Delete RAM:T/Temp
Echo "Done!"
```

## LISTING 2

```
/* *** backup.c
*******************************************

If you're using the Lattice/SAS compiler you can compile
and
link this code just typing:
   LC -b -r -v -Lcdn BackUp

Usage is:
   Backup From To [ALL] [QUIET]

***************************************************************
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <proto/dos.h>
#include <exec/memory.h>

#define OUT_OF_MEMORY      -1
#define EXAMINE_FAILURE    -2
#define CREATEDIR_FAILURE  -3
#define NEWLOCK_FAILURE    -4
#define COPY_FAILURE       -5

/* A few global variables */

BPTR sourcelock, oldsourcelock, destlock;
```

```
struct FileInfoBlock *fib;
BOOL all, quiet;

void string_to_lower(char *);
void output(char *);
void cleanup(char *, int);
long filecopy(BPTR, char *, SHORT);

int main(int argc, char *argv[])
  {
    long success;


    /* Check args: source directory and destination
directory
       are mandatory. That is, argc should be at least
equal
       to 3 */

    if (argc < 3)
        cleanup("Usage is: BackUp From To [ALL]
[QUIET]\n\n",
            RETURN_FAIL);
    /* Okay, are there any options in the command line? If
       so, first convert them to lower case, then check
them
       out */

    all = quiet = FALSE;

    if (argc > 3)
      {
        string_to_lower(argv[3]);
        string_to_lower(argv[4]);

        if ((strcmp(argv[3], "all") == 0) ||
            (strcmp(argv[4], "all") == 0))
          all = TRUE;
        if ((strcmp(argv[3], "quiet") == 0) ||
            (strcmp(argv[4], "quiet") == 0))
          quiet = TRUE;

        if (all == FALSE && quiet == FALSE)
            cleanup("Options must be ALL and/or QUIET\n\n",
                RETURN_FAIL);
      }


    /* Get a lock to the source directory and, if possible,
       make the source directory the current directory.
       P.S. Locking a file or a directory it's the quickest
       way to find out if such a file or directory exist!
*/

    sourcelock = Lock(argv[1], ACCESS_READ);
    if (sourcelock == NULL)
        cleanup("Can't find SourceDir!\n", RETURN_FAIL);
    oldsourcelock = CurrentDir(sourcelock);
```

```
/* So far so good, now let's allocate enough memory for
   a FileInfoBlock structure. A FileInfoBlock structure
   holds a lot of useful informations about a directory
   or a file and is listed in the include file
   libraries/dos.h */

fib = (struct FileInfoBlock *)AllocMem
        (sizeof(struct FileInfoBlock), MEMF_CLEAR);
if (fib == NULL)
    cleanup("Out of memory?!?\n", RETURN_FAIL);


/* Okay, now we can use the Examine() function to fill
   the FileInfoBlock with the information about the
   source directory */

success = Examine(sourcelock, fib);
if (success == 0)
    return("Can't examine SourceDir!\n", RETURN_FAIL);


/* Wait a minute! Is this really a directory? We better
   check it out... Within the FileInfoBlock structure
   there's an item called fib_DirEntryType: if it is
   lesser than 0 then we've locked a file! */

if (fib -> fib_DirEntryType < 0)
    cleanup("SourceDir is not really a directory!\n",
        RETURN_FAIL);


/* Okay, before we fall int the main loop, let's do the
   same for the destination directory */

destlock = Lock(argv[2], ACCESS_READ);
if (destlock == NULL)
    cleanup("Can't find DestDir!\n", RETURN_FAIL);

success = Examine(destlock, fib);
if (success == 0)
    return("Can't examine DestDir!\n", RETURN_FAIL);

if (fib -> fib_DirEntryType < 0)
    cleanup("DestDir is not a directory!\n",
        RETURN_FAIL);


/* Lucky guys!!! These really ARE directories... Now I
   guess it's time to enter the main loop... */

success = filecopy(sourcelock, argv[2], 0);

if (success < 0)
    cleanup("WARNING: Back-up might be incomplete!\n\n",
        RETURN_FAIL);
else
    cleanup("Done.\n\n", RETURN_OK);
}
```

```
/* This one is really trivial: just converts a string to
   lower case... */

void string_to_lower(char *string)
  {
  while(*string)
    {
    *string = tolower(*string);        string++;
    }
  }
/* And what about this one instead of printf()? We are
just
   printing strings, after all... */

void output(char *string)
  {
  Write(Output(), string, strlen(string));
  }
```

```
/* This one is called by main() when an error has occured
   or when execution has been completed. */

void cleanup(char *string, int error)
  {
  Write(Output(), string, strlen(string));

  if (destlock != NULL)
     UnLock(destlock);

  if (oldsourcelock != NULL)
     oldsourcelock = CurrentDir(oldsourcelock);

  if (sourcelock != NULL)
     UnLock(sourcelock);

  if (fib)
     FreeMem(fib, sizeof(struct FileInfoBlock));

  exit(error);
  }
```

```
/* This is the big one! This routine recursively calls
   itself until there are no more nested directories to
   examine. A few words about recursion?!?
   (...Deep breath...)
   When a function calls itself, new local variables and
   parameters are allocated on the stack and the function
   is re-executed from the start with these new variables.
   Once each recursive call returns, the old local
   variables and parameters are restored from the stack,
```

```
so
    execution can resume from the point of the function
call
    inside the function.
    (Pheeeewwwww!!!) */

long filecopy(BPTR lock, char *destdir, SHORT dirlevel)
  {
  struct FileInfoBlock *m;
  int size;
  long success;
  char *target = NULL;
  char *command = NULL;
  BPTR targetlock, dummylock, newlock;
  register SHORT j;



  size = sizeof(struct FileInfoBlock);
  m = (struct FileInfoBlock *)AllocMem(size, MEMF_CLEAR);
  if (m == NULL)
      return(OUT_OF_MEMORY);

  success = Examine(lock, m);
  if (success == 0)
      return(EXAMINE_FAILURE);


  /* Now we can use the ExNext() function to examine
     all the entries within the same directory.
     When ExNext() returns a 0 we know there are no more
     entries to examine... */

  while((ExNext(lock, m)) != 0)
    {
    if (m -> fib_DirEntryType > 0)    /* Directory? */
      {
        if (all == FALSE)    /* Skip nested directories?
*/
          {
          if (quiet == FALSE)
            {
              /* Tell about the directory being skipped
*/
                output(&m -> fib_FileName[0]);
                output(" (dir)..[skipped]\n");
            }
          continue;    /* and jump to the next entry */
          }

        else   /* Take into account nested directories */
          {
            target = (char *)AllocMem(256, MEMF_CLEAR);
          if (target == NULL)
            {
              FreeMem(m, size);
              return(OUT_OF_MEMORY);
            }
            strcpy(target, destdir);
```

```
  if (destdir[strlen(destdir) - 1] != ':')
    strcat(target, "/");
  strcat(target, &m -> fib_FileName[0]);

if (quiet == FALSE)
  {
    for (j = 0; j < dirlevel; j++)
      output("   ");
    output(&m -> fib_FileName[0]);
    output(" (dir)");
  }

/* If the target directory doesn't exists
   then we MUST create it! */

targetlock = Lock(target, ACCESS_READ);
if (targetlock == NULL)
  {
    if (quiet == FALSE)
      output("..");
    dummylock = CreateDir(target);
    if (dummylock == NULL)
      {
        if (quiet == FALSE)
            output("Not created!\n");
        FreeMem(target, 256);
        FreeMem(m, size);
        return(CREATEDIR_FAILURE);
      }
    if (quiet == FALSE)
        output("[created]\n");
    UnLock(dummylock);
  }

/* Otherwise we can continue... */

else
  {
```

The remaining code for
Listing Two can be found on
the AC's TECH Disk along with
all other files
necessary for this program.

Please write to:
Werther Pirani
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

# Programming The Amiga's GUI* in C

## *Graphical User Interface

*by Paul Castonguay*

This article is part of a regular series designed to help you take advantage of many of the custom features of your Amiga using the C programming language. In this issue you will find:

* A discussion of the Amiga's internal message system and how it can be used to control your program's operation from the mouse or keyboard.

* A discussion of the Amiga's font system.

* New functions added to the programming shell that we have developed together over the course of the last few issues.

* A correction to last issue's article, having to do with how memory is declared when using Intuition's DrawBorder() function.

* An explanation of the Line_Anim program from earlier issues, which uses Intuition's DrawBorder() function to create smooth animation. This issue it has been enhanced by using the Amiga's disk-based fonts.

By now you are beginning to develop a flavor for what it's like programming the Amiga's GUI. If I had to as succinctly as possible describe that experience, I would probably say that it was an exercise in data structures. That is to say, programming the Amiga is largely a matter of constructing certain data structures, connecting them together in some required way, and then calling various system routines that use them to accomplish a desired effect. As a result, your programming ability in this environment is dependent not only on your proficiency in C, but also on your familiarity with the wide variety of data structures and functions that make up the Amiga's run time libraries. And although C is oftentimes referred to as a transportable, system-independent language, what we are doing with it here is not transportable at all.

## PROGRAMMING SHELL, SHELL.C

Due to space limitations in this issue, copies of the programming shell could not be placed within each example directory. If you want to modify and re-compile any examples, you must first copy Shell.o and Shell.h from the Shell directory into the example directory. You can then either enter LMK on the AmigaDOS command line, or double click the Build icon on the Workbench. Since the shell is used at the object code level, it is not necessary to copy its source code, Shell.c, into example directories,.To start your own new projects using the programming shell, you must first create a project directory, as required by SAS/C. If you like to use icons, simply double click SAS/C's sascsetup icon (in the LC: logical device). It will prompt you for a directory name for your new project. If you prefer operating from the command line, use the AmigaDOS, MakDir command. Next copy into that directory Shell.o, Shell.h, lmkfile, and User_Program.c. Do not copy Shell.c. It is not required. User_Program.c is a convenient generic starting place for all your programs. For more information on the practical operational aspects of SAS/C, refer to earlier issues of this article series.

This programming shell is a convenient environment in which to conduct programming experiments on your Amiga. It is not a competitive, minimum size, development environment. Observant readers will notice that Shell.c is a rather large file, about 2000 lines. However, because you use its pre-compiled version, Shell.o, it does not affect the compile time of your projects. If you modify the Shell.c source code itself, and you want to re-compile it, use the make file called LMK_Shell.c, also located within the Shell directory.

# GOTTA GETTA GUIDE!

**L**ooking for a specific product for your Amiga but you don't know who makes it? Want a complete listing of all the Fred Fish software available? Just looking for a handy reference guide that's packed with all the best Amiga software and hardware on the market today?

If so, you need *AC's GUIDE to the Commodore Amiga*. Each *GUIDE* is filled with the latest up-to-date-information on products and services available for the Amiga. It lists public domain software, users' groups, vendors, and dealers. You won't find anything like it on the planet. And you can get it only from the people who bring you the best monthly resource for the Amiga, *Amazing Computing*.

So to get all this wonderful information, call **1-800-345-3360** today and talk to a friendly Customer Service Representitive about getting your *GUIDE*. Or, stop by your local dealer and demand your copy of the latest *AC's GUIDE to the Commodore Amiga.*

## MOVING?

## SUBSCRIPTION PROBLEMS?

Please don't forget to let us know. If you are having a problem with your subscription or if you are planning to move, please write to:

**Amazing Computing Subscription Questions**
**PiM Publications, Inc.**
**P.O. Box 869**
**FallRiver, MA 02722**

Please remember, we cannot mail your magazine if we do not know where you are.

Please allow four to six weeks for processing

# AC's TECH Disk
## Volume 2, Number 3

## A few notes before you dive into the disk!

• You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

• In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' (which is provided in the C: directory). lharc archive files have the filename extension .lzh.

To unarchive a file *foo.lzh*, type *lharc x foo*
For help with lharc, type *lharc ?*
*Also, files with 'lock' icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.*

AC's TECH DISK
GOES HERE!

Please notify your
retailer if the
AC's TECH Disk
is missing.

We pride ourselves in the quality of our print
and magnetic media publications. However, in
the highly unlikely event of a faulty or dam-
aged disk, please return the disk to *PiM
Publications, Inc.* for a free replacement.
Please return the disk to:

AC's TECH
Disk Replacement
P.O. Box 2140
Fall River, MA 02720-2140

*Be Sure to
Make a
Backup!*

## CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc, their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright ©1990, 1991 by PiM Publications, Inc, and may not be duplicated in any way. The purchaser however is encouraged to make an archive/backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work, twice, to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PiM Publications, or any of it's agents, is not responsible for any damages incurred while attempting this project.

Two more things: First, starting in this issue, I am compiling the Shell.o module, and all example programs, using SAS/C's absolute addressing option, -b0. Example sizes are getting large enough that you will start to have trouble linking them if you continue to use the relative addressing mode, -b1. Second, be warned that you *must* increase your system stack size over the Amiga's default 4000 bytes. I use 30,000 bytes on my system. For all the examples in this article 8000 bytes will work fine.

Now, let's get on with this issue's article.

## INPUT/OUTPUT PROGRAMMING

If you design a program to use the AmigaDOS command line to interact with the user, you can use the normal, Standard C scanf() function to make it respond to keyboard entries. But such programs are limited to simple text style operations, and are unable to take advantage of the real graphic power of the Amiga. To accomplish keyboard input within the realms of Intuition, you must use a different approach, and as you should be growing to expect, there are several ways of doing it. Some are high level, requiring relatively little knowledge of the internal workings of the machine. Others are lower level and require more. Naturally my first introduction to this will be at a level that requires the least amount of effort on your part. Thus you will be able to enjoy the benefits of programming your Amiga in C as quickly as possible.

## TELEPHONES AND COMPUTERS COMPARED

Think for a moment about your telephone and how through a single line connected to your home you are able to communicate with millions of different people around the world. Wonderful isn't it? All that is possible because your one telephone line is physically connected to a local, central dialing office (near your home) through which you can be switched or connected to a variety of high capacity, long distance circuits, along which your voice travels with thousands of others until it ultimately gets intercepted by another central dialing office situated close to the person you are calling. These high capacity circuits are called trunk lines. Thus all telephones are interconnected, not directly, but through a shared, communications system. Note also that although there may be a lot of activity on the telephone system at any one time, your phone doesn't ring until someone, somewhere, generates a message specifically for you. Computers have similar communication systems that provide message handling between their various peripherals and programs. The term "message stream" is used to describe the equivalent of the telephone company's trunk lines. Messages generated by different peripherals and programs travel back and forth along this common facility, each satisfying perhaps a different requirement. The term "message port" is used to describe the equivalent of the telephone company's central dialing office. These are locations where different programs and peripherals are given access to the message stream. For your program to receive mouse or keyboard information, it must somehow get connected to a message port, where it can be given access to the message stream along which such information travels. Note that although there may be a lot of activity on a computer's message stream at any one time, your program should not respond until somewhere a message is generated just for it.

## THE AMIGA'S INTERNAL MESSAGE STREAM

The easiest way to connect your program to the Amiga's message stream is to ask Intuition to do it for you, and you can do that during your program's initialization phase by assigning a special value to one of the members in its NewWindow structure. Recall that it is within this NewWindow structure that you enter
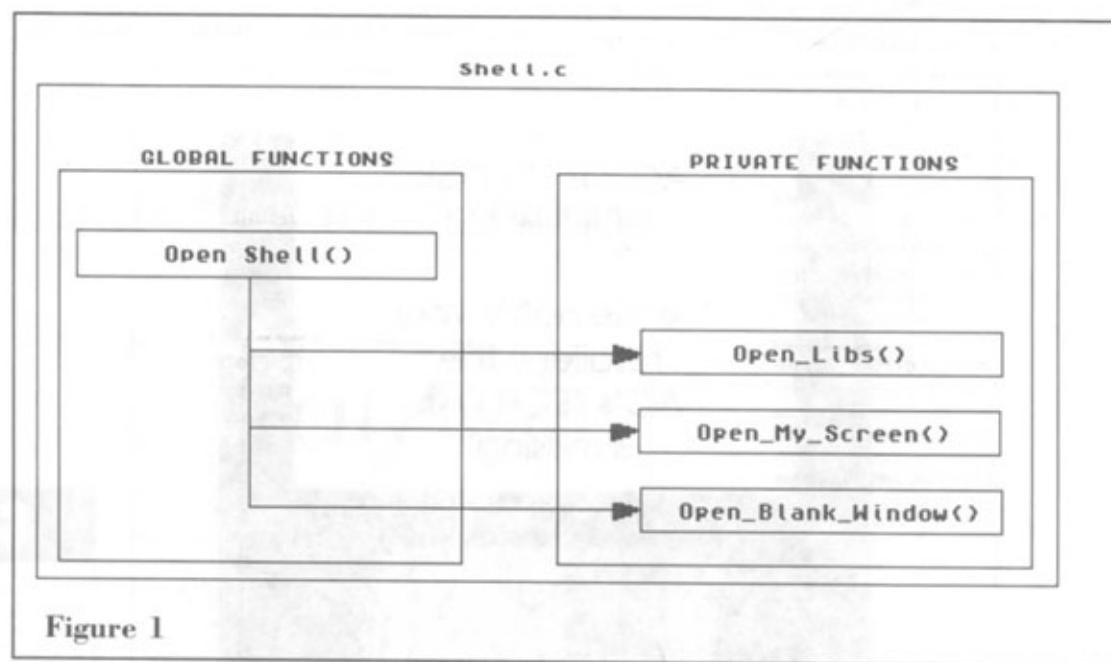


**Figure 1**

a description of the window you want Intuition to generate for your program. One of its members is called IDCMPFlags, and its purpose is to tell Intuition what kind of communication you want your program to have with the rest of the system. Intuition responds by setting up a message port which connects your program to the Amiga's message stream. That rather cryptic name, IDCMP, is an acronym for "Intuition Direct Communications Message Port." The actual values that you assign to IDCMPFlags are macros, which have been conveniently defined for you in the <intuition/intuition.h> header file. For example, to have your program receive messages from the mouse you assign the macro MOUSEBUTTONS.

## ADDING IDCMP TO Open_Shell()

In our programming shell, the NewWindow structure is hidden away within the Open_Blank_Window() function, a private function within Shell.c. Such concealing of the details of creating windows is a desirable way to design programs. It helps us to better come to grips with their complexity. This is an example of the formal, computer science principle called "information hiding," remember? But now we need to gain access to some of that hidden detail, so let's go in there and see how we can do that.

In figure #1 I show the hierarchical structure within the part of the shell that gives us our graphic drawing surface:

In article 3 of this series the prototype definition of Open_Blank_Window() was:

```
BOOL Open_Blank_Window(struct Screen *my_screen,
            ULONG my_idcmp,
            struct Window *(*my_window),
            struct RastPort *(*my_rp))
```

As you can see, I have already designed this function to accept an argument for assignment to the IDCMPFlags member of the NewWindow structure. Now isn't that convenient. Within the body of that function the parameter "my_idcmp" gets assigned to the IDCMPFlags member of the window_description, NewWindow structure. So to set up communications between your program and the mouse all you need to do is pass the macro MOUSEBUTTONS to the Open_Blank_Window() function.

Moving up the hierarchical tree of Shell.c (from article #3), we look within the Open_Shell() function and notice the following call:

```
Open_Blank_Window(*my_screen, NULL, my_window,
my_rp)
```

where a NULL is passed as the IDCMP argument. A NULL assigned to the IDCMPFlags member of the NewWindow structure tells Intuition not to set up a communication connection between the program and the system's message stream. In those early examples I wanted to keep things simple. In order to set up such communications now we could simply change that NULL to MOUSEBUTTONS, but that would not be correct. You see, the above call is within our Shell.c module, which we want to pre-compile and use "as is" in many different programming examples. Adding MOUSEBUTTONS at that point would force all our programs to use exactly the same IDCMPFlags definition. Instead we want to be able to specify MOUSEBUTTONS, and other possible macros, from the main program, when we call Open_Shell().

Here is the prototype definition of Open_Shell() as it stood last issue:

```
BOOL Open_Shell(struct Screen *(*my_screen),
    struct ViewPort *(*my_svp),
    struct Window *(*my_window),
    STRPTR resolution)
```

Too bad, I forgot to design that function to pass a value for the IDCMP message system. Well, we'll just have to add a new parameter to do that right now.

If you look at this article's source code for Shell.c you will see the new prototype definition:

```
BOOL Open_Shell(struct Screen
*(*my_screen),
            struct ViewPort *(*my_svp),
            struct Window *(*my_window),
            STRPTR resolution,
            ULONG my_idcmp);
```

as well as a modified Open_Shell() function which passes your IDCMP request on to Open_Blank_Window(). Now you can specify a particular IDCMP macro directly from main(), in the call to Open_Shell(). To create a window that will receive mouse click messages you make the following call:

```
Open_Shell(&my_screen, &my_svp, &my_window,
&my_rp, "LACEHIGH16", MOUSEBUTTONS);
```

## PROGRAM MAINTENANCE

I walked you slowly through the above modification process in order to make a point, that in a well structured, well documented program, it is easy to find the correct spot to add a new feature. Note, however, that the compiled code, Shell.o, of this article will not work with examples from previous issues because the number of parameters in the Open_Shell() function is now different. But because we have been diligent enough to use the new ANSI prototype definitions in all our code, the compiler should warn us if we ever accidentally try to compile an example with a wrong version of the programming shell..

## READING AND REPLYING TO MESSAGES

Having informed Intuition what kind of messages we want our program to receive, we must now consider how to read them once they arrive at its message port. Your first question is surely, "Where is that message port?" Its address is stored in the Window structure that Intuition created for you when your program invoked Intuition's OpenWindow() function. It is in a member called UserPort. In our programming shell we have been using the variable name "my_window" to point to our program's Window structure, so the address of its message port is "my_window->UserPort." Note that the UserPort itself is a structure of type "struct MsgPort," defined in <exec/ports.h>.

When messages arrive for your program, Intuition saves them in special buffers that it allocates dynamically, and it hangs on to them until you get a chance to read and process them. The messages themselves are stored according to a C structure called IntuiMessage, which is defined in the <intuition/intuition.h> header file. For the moment let's not worry about the details of that structure. Just be aware that it exists and that your messages are being stored dynamically, somewhere in memory, according to it. As more messages arrive, Intuition piles them up, dynamically allocating more memory space to hold them.

To read a message, you must invoke the GetMsg() function, using as an argument the address of your program's message port, in this case "my_window->UserPort.". The GetMsg() function returns the address of the first in a possible list of messages waiting to be processed by your program. Since messages are stored according to an IntuiMessage structure, we

assign the address returned by GetMsg() to a pointer of that type, like so:

```
struct IntuiMessage *my_message     my_message =
(struct IntuiMessage *)GetMsg(my_window-
>UserPort);
```

If there is no message in the port, GetMsg() returns a NULL.

Reading a message does not satisfy Intuition that you are finished with it. Intuition continues to hold on to messages until you specifically give it permission to do otherwise, and you do that by formally replying to them. Replying to a message confirms to Intuition that its contents are no longer needed and that it is OK to deallocate its temporary buffer. You reply to a message by invoking the ReplyMsg() function, using as an argument the address of the message you are finished with, like so:

```
ReplyMsg((struct Message *)my_message);
```

## WHEN DO MESSAGES ARRIVE?

There are two methods for checking if and when messages arrive at a window's message port. They are called waiting and polling. In the waiting method, your program pauses (stops normal execution) until a message arrives. This method is similar to using an INPUT statement in BASIC, where the program temporarily suspends all operations in anticipation of some user response. It is used whenever your program needs something critical for its operation, something without which it cannot proceed. In contrast, the polling method does not suspend your program's operation. Normal processing continues, while at the same time regularly checking the message port to see if any messages arrive. This method is similar to the KEY INPUT statement of True BASIC, or the INKEY$ statement of AmigaBASIC. It is used in simulation and animation programs where processing must not be interrupted, but where it is also necessary to respond to user input. The above example, First_Mouse.c, used this polling method to check for a mouse click by continually looping around a GetMsg() instruction. The example did not perform any other processing while it was

---

You can ask your program to receive both mouse and keyboard messages, by using C's bitwise inclusive OR operator between the MOUSEBUTTONS and VANILLAKEY macros.

---

The cast to (struct Message *), rather than (struct IntuiMessage *), is needed because that is the type defined for the argument of the ReplyMsg() function in its prototype definition, in the <proto/exec.h> header file. Struct Message is a more general structure used when you define your own message ports, as opposed to using ones generated for you by Intuition. But, of course, that's a subject for a future article.

The two functions, GetMsg() and ReplyMsg() are always used together. If you do not reply to messages, their buffers will not be de-allocated and your program will eat away memory as more and more of them arrive. Below I show a simple example of a program that reads a port until a message arrives. The "do loop" executes until the pointer "my_message" receives a legal address for a message. The example does not process the message; it simply acknowledges its receipt by replying immediately to it.

```
struct IntuiMessage *my_message
do          .
{
      my_message = (struct IntuiMessage
*)GetMsg(my_window->UserPort);
} while(my_message == NULL);
ReplyMsg((struct Message *)my_message);
```

You will find the above example on the magazine disk, in the drawer called First_Mouse. In that program the macro MOUSEBUTTONS is sent to the Open_Shell() function. Thus, pressing the left mouse button anywhere in the window causes execution to exit the "do loop" and the program to terminate.

polling, but of course it could have. Later in this article, you will see how the Line_Anim program uses polling to test for user response, while at the same time processing a complex graphic animation.

## WAITING FOR MESSAGES

To temporarily suspend your program's operation and have it wait for a message to arrive, you invoke the Wait() function. It takes a single argument, called an input mask, which allows the system to signal your program when a message arrives.

A mask is a number that is generally used to test for values at certain bit positions in binary numbers. For example, to find out if bit 3 of a binary number is set (equal to one) you could use C's bitwise AND operator with a mask, like this:

```
      binary equivalents
      ==================
              /* 76543210 <== bit positions */
d = 43;       /* 00101011 <== number to test */
mask = 1 << 3 /* 00001000 <== mask          */
if(d & mask)
      printf("Bit 3 is on\n");
else
      printf("Bit 3 is off\n");
```

The "&" character is C's bitwise AND operator. The "<<" is C's bitwise left shift operator. Refer to page 48 of *The C Programming Language*, Second Edition, by Kernighan & Ritchie, Prentice Hall 1988, if you need to brush up on C's bitwise operators.

The value of (d & mask) is zero for all values of d except those whose 3rd bit is set (bits are numbered from the right starting with 0). Thus the above code tests for the presence of the third bit in any number d. I used C's bitwise left shift operator to construct the mask, rather than the equivalent decimal number 8, because it identifies explicitly the bit position that the mask is testing for. Although writing (d & 8) would work exactly the same, someone reading the code, and trying to figure out how it worked, would have to recall that the number 8 represented bit position 3.

Well, that's what masks are used for, and although you do not need to know the exact details of how the operating system uses the particular mask that you pass as an argument to the Wait() function, you do need to know how to construct it. You do that in the same way as I constructed the above mask, by using C's bitwise left shift operator on the number 1. But how much should you shift it? That magic value is conveniently stored in your program's UserPort, in a member called mp_SigBit (message port signal bit). The mp_SigBit member of your program's UserPort contains a bit position that the system must use in order to signal your program when a message arrives. From that value you construct the mask, like so:

```
mask = 1 << my_window->UserPort->mp_SigBit;
```

You yourself need not know the bit position of that mask, but of course you can find out, if you want to, by placing a few printf() instructions in any one of my examples. Doing so will help solidify your understanding of what is going on here. Remember, however, that printf() instructions display text in the AmigaDOS window from which your program was launched, not in its graphic window. Therefore, if you want to add such scaffolding to your program, you should launch it from the AmigaDOS command line, as opposed to double-clicking its icon from the Workbench.

You call the Wait() function with the above mask, like this:

```
Wait (mask);
```

The result is that execution of your program halts—it sort of enters a state of suspended animation—until a message is received at its message port. Waiting is the preferred method of accepting keyboard or mouse response in a multitasking system because it allows the operating system to allocate more system resources to other tasks that may be running at the same time. You will find an example of the wait method of reading messages on the magazine disk, in the drawer called Waiting.

## INPUT FROM THE KEYBOARD

To have your program receive messages from the keyboard, you can use the IDCMP macro called VANILLAKEY. It causes your program to receive ASCII codes corresponding to whatever letters the user presses on the keyboard. You can also ask for your program to receive both mouse and keyboard messages, by using C's bitwise inclusive OR operator between the MOUSEBUTTONS and VANILLAKEY macros. The example in the directory called First_Keyboard() does that.

## STRUCTURED EVENT ENQUIRY

I would now like to design two new functions, to make it easier for us to find out if messages exist at any time in a program's

UserPort. One of these functions will use the wait method, putting program execution on stand-by until a message arrives. This function will be useful in many of our examples for gracefully terminating program operation. We can simply display a prompt like, "... press any key to quit ...", or "... click left mouse button to quit ...", and then wait for the user to respond. The other function will use the polling method, and it will be useful in animations, where processing must proceed while at the same time responding to user entries. Neither of these functions will process the received messages in any way. I am not interested at this point in what these messages mean, or even where they come from. I just want to know when they arrive, and of course I want to properly reply to them.

### Sleep() FUNCTION:

Below is my completed design for the first function, which uses the wait method of event enquiry:

```
VOID Sleep(struct Window *my_window)
{
        struct IntuiMessage *my_message = NULL;
        ULONG mask;

mask = 1 << my_window->UserPort->mp_SigBit;

Wait(mask); /* program execution stops here until
user responds */

while(my_message = (struct IntuiMessage
*)GetMsg(my_window->UserPort))
        ReplyMsg((struct Message *)my_message);
}
```

This function accepts as an argument the address of the Window structure of your program. It constructs the required input mask, and then calls the Wait() function. At that point program execution stops until the user performs some required response. Upon being revived, the function executes a "while-loop" to purge the message port of all other messages, in the event that more than one was received.

### UserPort_message() FUNCTION:

Below is my completed design for the second function, which uses the polling method of event enquiry:

```
BOOL UserPort_message(struct Window *my_window)
{
        struct IntuiMessage *my_message = NULL;
        my_message = (struct IntuiMessage
*)GetMsg(my_window->UserPort);
        if(my_message == NULL)
                return(FALSE);
else
{
        ReplyMsg((struct Message *)my_message);
        while(my_message = (struct IntuiMessage
*)GetMsg(my_window->UserPort))
        ReplyMsg((struct Message *)my_message);
        return(TRUE);
        }
}
```

This function also accepts as an argument the address of the Window structure of your program. It returns a TRUE or FALSE to report the presence or non-presence of a message in its UserPort. If a message exists, the function immediately replies to it, and then purges the port of any additional messages that may have been received.

The above two functions have been placed in Shell.c of this article. Program examples demonstrating their use are in the drawers called Polling and Sleeping. The polling example flashes a prompt on the screen, while simultaneously calling UserPort_message(). It does this by continually executing a "do loop" whose exit condition is:

```
while(!UserPort_message(my_window))
```

which means, "loop while there is no UserPort message for my_window."

In previous articles my examples would remain on screen for a fixed period of time, determined by the Delay() function. That was very crude. Now with IDCMP messages we can end our examples in a more professional manner.

There is a lot more to learn about IDCMP messages. I have not even mentioned what information they contain, nor how your programs can go about processing them. I will return to IDCMP's again, in a future article.

## FONTS

The Amiga has a standard set of fonts that can be used to enhance any program, allowing you to give your work a professional appearance. Here are the standard fonts available on all Amiga's:

| Version 1.3 | Version 2.04 |
| --- | --- |
| Courier | CGTimes |
| Diamond | GTriumvirate |
| Emerald | Courier |
| Garnet | Diamond |
| Helvetica | Emerald |
| Opal | Garnet |
| Ruby | Helvetica |
| Sapphire | LetterGothic |
| Times | Opal |
| Topaz | Ruby |
| Sapphire | Times |
| | Topaz |

On a version 1.3 Amiga the Courier, Helvetica, and Times fonts are located on the Extras disk. Each font styles comes in more than one size, adding up to a total of 32 sizes in 10 styles for version 1.3. For information on how to configure a floppy system such that all fonts are available, refer to "System Configuration Tips for SAS/C," Volume 1, Number 3 of *AC's TECH/AMIGA*. Topaz is the font style contained in the Amiga's KickStart ROM, and it is usually referred to as the Amiga's default font. It comes in two sizes, selectable on a version 1.3 system from the preferences screen. All other fonts are disk based. That is, their graphic data exists on disk. For some programs, like full-featured editors and word processors, it is important that they use whatever font the user has selected as a system default.

But as a hobbyist you may not want to design that feature into your own programs, at least not for a while. Designing programs to work elegantly using different default fonts can be tedious. Chances are you will want to move quickly to more interesting programming exercises. Thus, a good practical approach is to have your program ignore whatever font the user has selected as a system default, and to load instead whatever ones



&TextAttr ▶ OpenFont () ◀▶ TextFont

Legal?
No → take corrective action
Yes → proceed
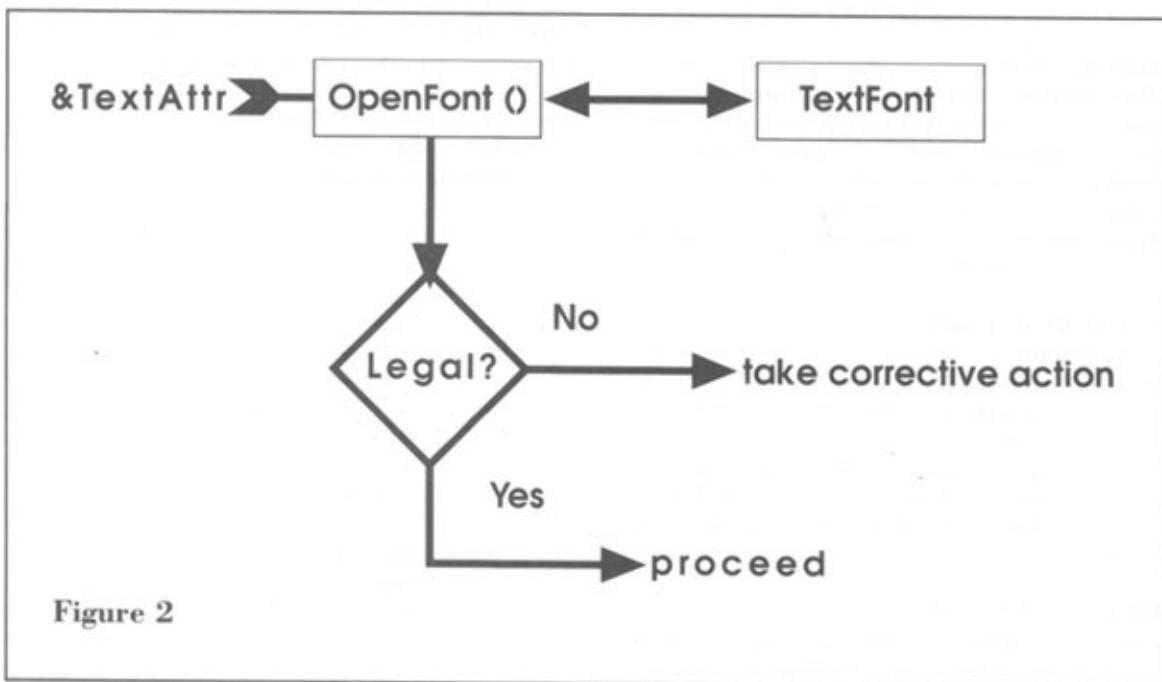
**Figure 2**

you decide it should use. Even if all you ever use is topaz, 80-column text, there is no real guarantee that your friends with whom you share your work will indeed select that same font as a system default when they run your program, especially now with the new 2.0 operating system. To be sure that your program gets the font, or fonts, that it needs, it should explicitly pick them. This does not mean that you no longer need to account for different sized fonts, but by your making the decision of which ones to use, your program's design is greatly simplified.

## OPERATIONAL STEPS

To use a particular font within your program, you must use the following operational plan:

1 - Declare a TextAttr structure in which you will store a description of the font you wish to use. I like to call this the description structure. This is not the font itself, only a description of it.

2 - Declare a TextFont pointer that will eventually hold the address of the font you wish to use. This is not an instance of the font, but a pointer to one. It is usually initialized to NULL.

3 - Open the "diskfont.library" run-time library. Recall that the Open_Libs() function of our programming shell already does that for you.

4 - Assign to each member of the description structure the characteristics of the font you want to use.

5 - Call the system function that makes the font available, either OpenFont() or OpenDiskFont(), using as an argument the address of the description structure declared in step 1, and assign the value returned by the function to the pointer declared in step 2. Is this getting to sound familiar?

6 - Test the value assigned to the font pointer to determine if it is indeed legally available. You will see either a legal address, representing a legitimate font that you can use, or a NULL, if for some reason the system could not make it available.

7 - Use the font pointer throughout your program to access its features and control it in different ways.

8 - Close the font before your program terminates, allowing the system to re-allocated whatever memory it was using.

## IMPORTANT STRUCTURES

There are two structures that are important when dealing with the Amiga's fonts, TextFont and TextAttr. TextFont is the most important of these. It is within a TextFont structure that the OpenFont() function loads a font into memory, making it available for use. It contains the actual bit character data of a font. For your program to use a font there must exist, somewhere in memory, a TextFont structure representing it, and of course your program must have a pointer to that structure. The TextAttr structure contains only a short description of a font. It is used in conjunction with the OpenFont() and OpenDiskFont() functions to load new fonts into memory, which is how you make them available. It is also used in conjunction with the IntuiText structure, to specify different fonts when using Intuition's high level, text rendering PrintIText() function, as well as in conjunction with the AskFont() function, to identify the current font, should you ever need to do that. The templates for both the TextFont and the TextAttr structures are in the <graphics/text.h> header file.

## THE AMIGA'S ROM BASED FONT

There are two sizes of topaz font stored in the KickStart ROM. They are often referred to by the number of columns that they can display on a high resolution screen; thus the names TOPAZ_EIGHTY and TOPAZ_SIXTY in the <intuition/preferences.h> header file. However, I will refer to them using names that reflect their pixel height, topaz8 for TOPAZ_EIGHTY, and topaz9 for TOPAZ_SIXTY.

To make a particular ROM-based font available to your program you must invoke the OpenFont() function. It takes one argument, the address of a TextAttr structure containing its description. You should have no trouble identifying the points from the above operational plan within the following code, which loads and uses topaz8:

```
struct TextAttr desired_font;/* step one of plan*/
struct TextFont *topaz8 = NULL;/* step two of plan
*/

desired_font.ta_Name = "topaz.font"; /* step four
of plan */
desired_font.ta_YSize = TOPAZ_EIGHTY;
desired_font.ta_Style = FS_NORMAL;
desired_font.ta_Flags = FPF_DESIGNED |
FPF_ROMFONT;
topaz8 = OpenFont(&desired_font);/* step five of
plan
*/
if(topaz8 == NULL);   /* step six of plan */
{
...take corrective action ...
}
SetFont(my_rp, topaz8);/* step seven of plan */
Move(my_rp, 0, 10);
Text(my_rp, "Hello", 5);

if(topaz8)                /* step eight of plan */
      CloseFont(topaz8);
```

The ta_Name member of the TextAttr structure is assigned a string consisting of the name of the font, a period, and then the word "font," all concatenated together. The ta_YSize member is set to the vertical pixel size of the font. I used the convenient macro TOPAZ_EIGHTY, from the <intuition/preferences.h> header file. The ta_Style member is set to normal, meaning that the font has no special style characteristics, like bold, italics, or underline. All the Amiga's standard fonts are stored as FS_NORMAL, although they can be algorithmically modified to create other styles—more about that in a future issue. The ta_Flags member is set to the font's location, in this case the KickStart ROM. The other macro, FPF_DESIGNED, is supposed to instruct the system to load a font that has exactly the characteristics you specify, although to tell you the truth I have not been able to verify that it works as intended. Perhaps I have been ill-informed on its purpose. Can anyone out there help?

The above code opens topaz8 for use in your program. If the user has already selected that font as the system default, the OpenFont() function will return its existing address; that is, it will return the address of the TextFont structure already representing that font in memory. Otherwise the font will be loaded into memory from the KickStart ROM. You should always verify that the address you receive from the OpenFont() function is legitimate, even for the ubiquitous topaz8. Opening fonts within your

program also gives you a new responsibility, that of closing them before your program terminates. You close previously opened fonts by invoking the CloseFont() function.

## USING A FONT

Opening a font places it in memory where it becomes available for use, but to actually use it, with the primitive Text() function, you have to tell the system to make it your program's current font. You do that with the SetFont() function, like so:

```
SetFont(my_rp, topaz8);
```

In contrast, the higher level PrintIText() method of rendering text does not require the use of the SetFont() function. It relies instead on a pointer to a TextAttr structure describing a previously loaded font. To render screens consisting of text in more than one font you construct multiple IntuiText structures, each one pointing to its own font TextAttr structure and text. You then link these structures together using their NextIText members, and render them by invoking a single PrintIText() instruction. The line animation program later in this article contains an example of this.

## POSITIONING TEXT

On the magazine disk, in the ROM_Fonts drawer, you will find an example that opens, uses, and finally closes the two ROM-based fonts. The example also uses a resolution independent technique for placing text on the screen. When using different fonts it becomes increasingly important to take into consideration their different sizes and baseline positions. You can read that important information from your window's RastPort structure, in the TxHeight, TxWidth, and TxBaseline members. Notice how I center text on the screen in the ROM_Fonts.c example:

```
stccpy((STRPTR)font_title, "topaz 80:",80);
x = (my_window->Width -
strlen((STRPTR)font_title) * my_rp->TxWidth)/2;
y = line_position;
Move(my_rp, x, y);
Text(my_rp, (STRPTR)font_title,
strlen((STRPTR)font_title));
```

The horizontal pixel position is calculated by using information stored on the system. The my_window->Width is the pixel width of the screen. The my_rp->TxWidth is the pixel width of one character in the current font. The calculation subtracts the pixel length of the text from the width of the screen and divides by two.

In the above example the vertical position is kept in a variable called "line_position". It is initialized using screen dimensions stored on the system:

```
line_position = my_window->Height/4;
```

and updated by incrementing it by the sum of the height of the font and a space of two pixels.

```
y = line_position += my_rp->TxHeight + 2
```

The above text positioning technique has the advantage that it is independent of the resolution of the screen. Try it out yourself.

Change the resolution to LACEHIGH2, and you will see that the text is still properly centered.

## DISK-BASED FONTS

To make a disk based font available to your program you must invoke the OpenDiskFont() function. It works exactly like OpenFont(). It accepts as an argument the address of a TextAttr structure, and it returns to your program the address of the corresponding TextFont structure that it loads into memory.

```
struct TextFont *diamond20;
struct TextAttr desired_font;

desired_font.ta_Name = "diamond.font";
desired_font.ta_YSize = 20;
desired_font.ta_Style = FS_NORMAL;
desired_font.ta_Flags = FPF_DESIGNED |
FPF_DISKFONT

diamond20 = OpenDiskFont(&desired_font);

if(diamond20 == NULL);
{
...corrective action...
}
```

The ta_Name member of the text attribute structure is constructed in the same way as for our earlier topaz example. The name "diamond.font" actually corresponds to a file on your system disk, in the FONTS: directory. Thus you don't need to spend time flipping through documentation to find the correct name for a font, you just look on your disk using the following command:

```
dir Fonts:
```

You will see that for each ".font" file there exists a corresponding directory in which the font's bit data is stored. To find out what sizes of diamond font are available, enter:

```
dir Fonts:diamond
```

You will see that the name of each "bit data file" is its actual size. On the magazine disk, in the drawer Disk_Fonts, you will find an example that loads into memory and uses the two available sizes of diamond font. It uses a new function for positioning text on the screen, TextLength(). You see, diamond is a proportional font. That is, its individual letters are not all the same width. In fact, that is the reason why it looks so nice. But the problem is, the value stored on the system for character width, my_rp->TxWidth, is now only an average. Using it to center text will give erroneous results. The TextLength() function solves that problem by returning the exact pixel length of any string in the current font.

```
SetFont(my_rp, diamond20);
length = TextLength(my_rp, "Hello", 5);
```

TextLength requires three arguments: the RastPort pointer, the string whose pixel length you want to measure, and the character length of the string. Here is one of the text centering calculations

from Disk_Font.c:

```
x = (my_window->Width - TextLength(my_rp,
line_one, strlen(line_one)))/2
```

It is important that you understand these text centering calculations.

## TOO MUCH DETAIL

If you are at all like me, you are beginning to get overwhelmed by the large amount of detailed work that must be done in order to use a particular font. Shouldn't things be easier? Of course. Practically speaking, if you intend to make regular use of the Amiga's fonts, you should organize this entire operation into one, simplified, structured form. But of course here is where many programmers differ. Everyone has their own programming style. Despite that, I have designed into our programming shell a convenient way for you to use the Amiga's fonts. I hope you like it. And if you don't, I hope it at least gives you some constructive ideas on how to design your own.

## GLOBAL FONT POINTERS

To allow you to conveniently use any font from any section of your program, I have decided to make their pointers global. Yes, I know it's dangerous to use global variables, but I promise to be good and document them well. Besides, my naming convention will make them easy to recognize. Each one will consist of the name of a font and its size concatenated together, as in sapphire19, or emerald17. You will find such global font pointers declared at the top of this issue's Shell.c file, one for each standard font style and size. You will also see some corresponding TestAttr structures, with names like ta_sapphire19, and ta_emerald17. All these global variables are referenced again within the Shell.h header file, as indeed they should be. Remember, it is in the Shell.h header file that your program picks up whatever global function and variable declarations it needs in order to use the Shell.o pre-compiled module.

## Load_Font() FUNCTION

I have designed this function to facilitate the loading of any font. It takes one argument, the name of the font you want to load, enclosed within quotes - like this:

```
error = Load_Font("sapphire19");
```

The function uses the string argument to identify which font you want. It then calls Intuition's OpenDiskFont(), using as an argument the address of the desired font's global TextAttr structure. It also assigns the address returned by OpenDiskFont() to the font's corresponding global pointer. Finally it returns a TRUE to signify successful completion. If at any point there is a problem, the function will return a FALSE. You can use the following popular calling style to invoke the Load_Font() function:

```
if(!Load_Font("sapphire19"))
{
...take corrective action ...
}
```

To actually use a font with the primitive Text() function you must first use the SetFont() function, to make it your program's current font:

```
SetFont(my_rp, sapphire19);
```

You should inspect the Load_Font() function in Shell.c to convince yourself that I have designed it correctly. SAS/C's strcmp() function is used to test the argument passed from the calling program. That's the same method I used in the Open_My_Screen() function to determine what screen resolution you wanted for your program. Notice that after opening a font I test its size. Intuition has the characteristic that it will give you a font that closely matches the one you asked for if it cannot find an exact match. I don't want that feature. Finally, at the end of the Load_Font() function you will see a lone "return FALSE" instruction. The purpose of that is to signal when a match could not be found for the string argument passed by the calling program, probably because it was misspelled.

## AUTOMATIC FONT CLOSING

The required closing of fonts has been designed into the Close_Shell() function. *It happens automatically!* Take a look inside this issue's Close_Shell() function to see how each global font pointer is tested before it is closed. An empty font contains a NULL and won't get closed. Note that passing a NULL to Intuition's CloseFont() function will crash your machine. That is in fact the very reason why I designed the Close_Shell() function to automatically close fonts for you, rather than ask you to do it yourself. When you use a lot of different fonts, it is easy to get mixed up and accidentally close one that was never opened, thus crashing your machine. By never closing fonts yourself, that is, by relying on the Close_Shell() function to do it for you, you will never encounter that problem. It is always best to hide dangerous operations within functions.

You will find an example that uses my structured method of accessing fonts in the Font_Function drawer on the magazine disk. You should agree that this new structured method is easier to use than resorting to basic principles every time.

## 2.04's OUTLINE FONTS

The new operating system contains three new fonts styles whose sizes are adjustable over a very wide range. They are called outline fonts, and they are stored on disk in a format that is different from the older, bit mapped fonts. However, Commodore supplies a system tool, called Fountain, which you can use to convert outline fonts into standard Amiga, bit-mapped format. For example, using the Fountain you can produce a bit-mapped file for a 100-point Compugraphic Times font, and you can use that font from within your programs using the same techniques presented here in this article. Naturally if you want to share that program with friends you will have to include with it a copy of whatever font files it uses.

## ERROR FROM LAST ISSUE

In the last issue I gave you some erroneous information having to do with allocation of memory for a border-data-array. Recall that the purpose of this array is to store pixel numbers representing points in a line drawing, and that it must be linked to a Border structure through the XY member. I told you last issue

that a border-data-array had to be declared in chip RAM, but I was wrong. You do not need to specify any particular type of memory for a border-data-array.

Recall that there are two methods for allocating memory for an array, the static method and the dynamic method. The static method is used when you know in advance the exact size that you will need, when you know the number of points in your line drawing. I said last issue that this static method required you to declare the border-data-array using SAS/C's special, chip data type, and that as a result it needed to be global in scope. That is not true. You can declare the border-data-array the same way as you do any other array, and as a result it need not be global. Note however that its scope must be such that it actually exists at the time the DrawBorder() function is invoked. The dynamic method of allocating memory for an array is used when you do not know in advance the exact size that you will need, when the number of points in your line drawing must in fact be calculated by the program itself. This method involves the use of AllocMem(), in which you specify, using variables, the amount and type of memory that you need. I said last issue that you had to use the macro MEMF_CHIP as an argument to the AllocMem() function. That is not true. You need not specify the type of memory at all. Thus, using an example from last issue, the following declaration would work fine:

```
Rosette_Data =
(SHORT*)AllocMem(2*(line_count+POINTS)*sizeof(SHORT),
            MEMF_CLEAR);
```

My errors last month do not affect whether or not you need to use the static or the dynamic method of allocating an array. They affect only the type of memory that you specify in each case.

It turns out that there are many other data structures on the Amiga that do require data to be declared in chip RAM. I will be presenting one next issue. As a result it is easy to forget and accidentally specify chip ram when in fact you do not have to, as I did last issue. It is not the first time that I have made this mistake, and it probably won't be the last. The good news is that this error does not affect the operational integrity of your programs. Specifying chip RAM when in fact you do not have to is not harmful. I do however want to set the record straight on this subject. In contrast, forgetting to specify chip RAM when in fact you need to can be disastrous. I hope I do not get caught making that mistake in these articles.

## LINE ANIMATION PROGRAM

This program was first introduced several articles ago for the purpose of giving you some valuable experience compiling multiple module projects using SAS/C. This issue I present its working parts, its algorithm. To facilitate its presentation, I have ported it to the programming shell that we have been developing together since that time. I have also enhanced it by using our newly learned skill, disk-based fonts.

## CREATING A LINE WITH A TRAIL

To create a line with a trail you draw several instances of a line, each one differing slightly in position. That is, you draw a group of lines that are spread out in a closely related, sequential pattern. In my program I draw 19 lines. To create a moving line with a trail, you must continue to draw new lines, while at the same time erasing older ones. The number of lines appearing on the screen must remain constant while you do this. To give the appearance of forward motion you must add lines to one end of the pattern and erase them from the other end, and to do that you must keep track of the positions of all the lines in the pattern.

## USING AN ARRAY FOR COORDINATES

An obvious way to keep track of the positions of lines is to store the coordinates of their end points in an array. Using index variables you can keep track of which line you drew first and which one you drew last. To create continuous motion you will want to erase the line at the trailing edge of the pattern and replace its coordinates in the array with a new one that is geometrically closely related to the one at the leading edge. For example, suppose I have a 19 element array containing the x and y coordinate positions of the first 19 lines that I drew. The one I drew first is at index position 0. It represents the line at the trailing edge of the pattern. The one I drew most recently is at position 18. It represents the one at the leading edge. I might start by erasing the line at position 0, recalculating a new one based on the position of the one at index position 18, storing its coordinates in the array at position 0 (overwriting the old one), and then finally drawing the line. The line at index position 0 would now represent the new leading edge of the pattern. I could then go to index position 1 of the array, and again erase, recalculate (based this time on the position of the line stored at index position 0), store, and draw. I might continue in this manner until I reach index position 18, the top of the array. At that point I simply jump back down to position 0 and start all over again. Thus I cycle through the array, each time erasing old lines, and recalculating and drawing new ones. I can use C's modulus operator to cycle through an array. For example, in the following code the 20 element array is cycled five times.

```
short array[20];

for(index = 0; index < 100; index++)
printf("%d\n", array[index % 20]);
```

When index == 20, then index % 20 == 0. Thus when the index reaches the top of the array, the subscript [index % 20] jumps back down to the bottom.

## CIRCULAR QUEUE

There is an official name for an array used as I have described. It's called a circular queue. The name queue is used to describe any structure in which data is processed on a first-in-first-out basis, or FIFO. In the above example the line that is erased (taken out) is always the oldest one, the one that has been in the array the longest, the one that compared to all the others got there first. The queue is called circular because of the way I cycle through it, using the modulus operator. When I increment from the the last position I automatically jump back down to the first. Although in reality every array consists of a linear section of memory, using it this way makes it seem like it's doubled back on itself, like a donut. Hence the name circular queue.

## DRAWBORDER() FOR DRAWING AND ERASING

Intuition's Border structure has some nice advantages for implementing the line animation program. I can declare an array of Border structures where each one draws one line, similar to

how I drew the rosette pattern last issue. Each Border structure is linked to four elements of an array which contains the coordinates of all the lines in the pattern. I call this the Border-data-array. I can then link these structures together and render them all at once using a single DrawBorder() instruction. In addition, since each Border structure specifies its own particular pen number, I can erase lines simply by specifying pen number zero for the one containing the line I want to erase. Note that in my program I use a 20-element array from which I draw a 19-line pattern. The extra element is used to automatically erase old lines as new ones are added.

In most implementations of this line animation program the erasing of the last line overwrites other lines as the pattern moves around screen, creating ugly spots. To remove this effect, you must redraw all lines in the pattern immediately after each old line is erased. With Intuition's DrawBorder() function this is easy to do. Recall that a linked list of Border structures is rendered sequentially, that is, the first one in the list is rendered first, then the second, etc. So, to guarantee that the oldest line in the pattern is erased first, followed by redrawing all the others, I permanently set the FrontPen member of the first Border structure in the linked list to the background color, zero, and then copy the coordinates of the old line that I want to erase into the first four positions of the Border-data-array. Thus the old line is linked to the Border structure whose FrontPen is set to zero. With a single DrawBorder() instruction the computer erases the old line then draws every other line in the pattern. On most computers drawing so many lines would bog the system down so much that the animation would be ruined. Not so on the Amiga, because of the speed of DrawBorder(). Below I show the function Init_Lines(), which accomplishes the initialization of the Border structures.

```
VOID Init_Lines(struct Border Lines[], SHORT
*points)
{
BYTE i;
/* ====================================== */
/* === Initialize Border structures === */
/* ===     and Link them together     === */
/* ====================================== */
for(i = 0; i < 20; ++i)
{
        Lines[i].LeftEdge = 0;
        Lines[i].TopEdge   = 0;

        if(i == 0)
                Lines[i].FrontPen = 0;
        else
                Lines[i].FrontPen = 1;
                Lines[i].BackPen  = 0;
        Lines[i].DrawMode = JAM1;
        Lines[i].Count    = 2;
        Lines[i].XY       = points + i*4;

        if(i != 19)
                Lines[i].NextBorder = &Lines[i+1];
        else
                Lines[i].NextBorder = NULL;
        }
}
```

FrontPen of the first Border structure is set to zero. It will do the erasing. All other structures in the list have their FrontPen members set to one. Each Border structure is linked through the NextBorder member, except for the last which is grounded (assigned a NULL). In addition each Border structure is linked to its own section of the Border-data-array, in which coordinates for the various lines will be stored. The first Border structure is linked to the first four elements of the array. Refer to page 59 of last issue for a hierarchical diagram of an equivalent structure, which was used to draw the rosette.

## COORDINATE CALCULATIONS

The calculations of the coordinates for the endpoints of each line are based on a trick of trigonometry. Recall that a circle can be drawn by using the SINE and COSINE functions. Look at the example on disk called Circle. Now, imagine two points travelling around on the circumference of that circle. Also imagine that those two points are connected by a straight line. The result would be a revolving straight line, like a fan spinning around.

If you change the relative frequencies and phase angle for x and y in the above program you get what is called a Lissajous figure. Look at the example on disk called Lissajous. You will see that the x angle has been multiplied by 3, and the y by 5. Also a phase angle has been added to the y angle. You can create all kinds of interesting patterns just by playing with the frequencies and phase angle in this program. Now imagine two points travelling around on the circumference of this curve, and a straight line joined between them. This is in fact exactly how the line motion of my program is calculated. There are other ways to calculate the motion of a line, some creating more realistic motion. This method is convenient.

## SPEEDING UP TRIG FUNCTIONS

Trigonometric functions are very time intensive on any computer, requiring the use of floating point libraries. That would slow down an animation on any computer. To solve that problem, I create a look-up table of trig values for sin() and cos(). Note that I use the names Sin and Cos (first letter capitalized) for these tables. They are integer arrays and I store in them the calculated pixel values corresponding to the sin() and cos() functions from the floating point library. I do this for all angles in one degree increments. Thus when my code uses Sin[] or Cos[], it is not calling functions in the slow math library, but integer values in a faster look-up table. Take a look at the code which generates these look-up tables:

```
s.xmin = -1.0;
s.xmax = 1.0;
s.ymin = -1.0;
s.ymax = 1.0;

if(!Scale_Window(my_window))
{
...corrective action....
}
for(angle = 0; angle < 360; ++angle)
{
        Sin[angle] = Fy(sin(Rad((DOUBLE)angle)));
        Cos[angle] = Fx(cos(Rad((DOUBLE)angle)));
}
```

Please notice that I have scaled the screen to +/- 1.0 in both the horizontal and vertical directions, and used Fx() and Fy() pixel conversion functions for calculating values at each angle. Thus the values stored in the look up tables represent horizontal and vertical pixel coordinates for a point rotating around a unit circle which fills the screen. Actually, it is an ellipse. Without screen scaling and pixel conversion functions, I don't know how I would do this.

## COLORS

I use C's bitwise AND operator to select a range of random numbers for color selection:

SetRGB4(my_svp, 1, 8+(rand()&7), 8+(rand()&7), 8+(rand()%7));
SetRGB4(my_svp, 2, 8+(rand()&7), 8+(rand()&7), 8+(rand()%7));
SetRGB4(my_svp, 3, 8+(rand()&7), 8+(rand()&7), 8+(rand()%7));

By choosing random values between zero and 7, and then adding 8, I select the brightest half of the Amiga's 4096 colors.

## PHASE ANGLES

I then select phase angles for both the horizontal and vertical coordinates of each endpoint of a line.

phase_x1 = 2+(rand()%6);
phase_y1 = 2+(rand()%6);
phase_x2 = 2+(rand()%6); phase_y2 = 2+(rand()%6);

Although I use the variable name phase, what I really mean here is an angular index value. The angular measure of the x and y coordinates of each endpoint of the line will be indexed using different values, each between 2 and 7 degrees, thus giving them constantly changing value and relative phase. As a result the two points travel around the circumference of the Lissajous figure at different speeds. This creates more variety. Next I randomly pick the initial angles, angle_x1 through angle_y2:

angle_x1 = rand()%360;
angle_x2 = rand()%360;
angle_y1 = rand()%360;
angle_x2 = rand()%360;

## LINE INITIALIZATION LOOP

Now for the meat of the algorithm. I place into the Border-data-array the coordinate values of 19 lines. Each end point has two coordinates, a horizontal and a vertical. Thus it takes four elements of the array to store one line. Notice that I start the first line at

points[4], not points[0]. Remember that the first four elements are linked to the Border structure whose FrontPen member is assigned zero. It is used for erasing, not for drawing. The lines calculated by the following loop are closely related geometrically because each one uses the next incremented value of its angle variable. Each variable cycles through 360 degrees using its own constant step value. The result is the coordinates of two points moving around the circumference of a Lissajous figure at different, but constant, speeds.

```
for(i = 4; i < 80; i += 4)
{
points[i]     = Cos[(angle_x1 += phase_x1) % 360];
points[i+1]   = Sin[(angle_y1 += phase_y1) % 360];
points[i+2]   = Cos[(angle_x2 += phase_x2) % 360];
points[i+3]   = Sin[(angle_y2 += phase_y2) % 360];
}
```

**Figure 3**



Name (0)

ta_sapphire19

FrontPen = 1
BackPen = 0
DrawMode = JAM1
LeftEdge = x
TopEdge = y
ITextFont ———
IText ———
NextText

ta_Name = "sapphire.font"
ta_YSize = 19
ta_Style = FS_NORMAL
ta_Flags =FPF_DISKFONT

"Line Animation"

Name (1)

ta_garnet16

FrontPen = 1
BackPen = 0
DrawMode = JAM1
LeftEdge = x
TopEdge = y
ITextFont ———
IText ———
NextText

ta_Name = "garnet.font"
ta_YSize = 16
ta_Style = FS_NORMAL
ta_Flags = FPF_DISKFONT

"using Intuition's DrawBorder()"

. . . ( last structure in list points to NULL )

## ANIMATION LOOP

And here is the main animation loop:

```
for(i = 0; i < 3000; i+=4)
{
/*
=====================================================
*/
/* === Copy data from position i%76 to position 0
=== */
/* === then put new data into position i%76 === */
/*
=====================================================
*/
points[0]          = points[i%76+4];
points[i%76+4] = Cos[(angle_x1 += phase_x1) % 360];

points[1]          = points[i%76+5];
points[i%76+5] = Sin[(angle_y1 += phase_y1) % 360];

points[2]          = points[i%76+6];
points[i%76+6] = Cos[(angle_x2 += phase_x2) % 360];

points[3]          = points[i%76+7];
points[i%76+7] = Sin[(angle_y2 += phase_y2) % 360];

DrawBorder(my_rp, Lines, 0, 0);

if(UserPort_message(my_window))
        goto GET_OUT;
}
```

Each time through the loop four coordinates are copied starting from the current position in the queue, (i%76+4), which represents the oldest line in the queue, into the first four elements in the Border-data-array, where it will be erased. Next the coordinates starting at that same current position, (i%76+4), are replaced by new values, calculated by incrementing forward the angle variables. It is this incrementing of the angle variables that causes the positions of new lines to be closely related geometrically, thus producing a smooth flowing pattern. Finally DrawBorder() is invoked to both erase the line whose coordinates were copied into the first position, and to draw the remaining 19 lines, including the most recently calculated one. The line most recently calculated (at the present queue position) appears at the leading edge of the pattern. The drawing process occurs so fast that you never notice that the lines are actually being drawn out of sequence.

The last instruction in the loop is a call to UserPort_message(), our newly designed polling function. It allows your program to process the animation while at the same time check for keyboard response, indicating that the user wants to terminate the animation. The animation loop executes 3000 times, then drops out to re-calculate initial angles and phase angles, thus creating a different pattern.

## IntuiText STRUCTURES AND FONTS

In the Init_Title() function of Line_Anim.c, you will see how fonts are used with the higher level PintIText() function. Recall that PintIText() renders text which has been previously linked to any number of linked IntuiText structures. To get different fonts,

each IntuiText structure is linked to a different TextAttr structure through its ITextFont member (Figure#3).

The calculations for centering text on the screen are done in the same manner as in our earlier Disk_Fonts.c example. Notice that to calculate horizontal screen positions I had to use the TextLength() function, and to do that I had to first change the current font using the SetFont() function. The LeftEdge member of each IntuiText structure is assigned a calculated horizontal position for its particular string. Perhaps you think it a bit tedious to have to perform all these calculations. The good news is that using this higher level PrintIText() method, as opposed to the more primitive Text() method, you have to perform these calculations only once. The main program can render the text as many times as it wants, without having to perform any additional calculations. The text appears properly centered every time because the horizontal and vertical positions of each text string have been pre-calculated and stored in the LeftEdge and TopEdge members of their respective IntuiText structures.

## RESOLUTION INDEPENDENCE

There are two versions of the compiled line animation program on disk. The only difference between their source code is the graphic mode specified in the call to Open_Shell(). This is possible because I have used system values in my calculations of text positions, and the Fx() and Fy() pixel conversion functions for the positions of lines. The program is resolution independent.

## WHAT'S NEXT

In the next issue I will present a different graphic rendering function, DrawImage(). We will be leaving the Rosette and Line Animation examples behind us, and going on to a new example, the game of *LIFE*. This game is really a simulation invented by British mathematician John Horton Conway. It is not only filled with intrigue, but also gives me an opportunity to present some interesting and challenging programming concepts. You will see that Intuition's DrawImage() function allows you to easily design this and other complex board games. I hope to see you next issue.

☑

# CAD *Application Design*

## PART 4: ADDING NEW OBJECTS AND FUNCTIONALITY

*by FOREST W. ARNOLD*

## Introduction

In Part 3 of this series (*AC's Tech*, V2.1), we saw how to use C programming and software packaging techniques to implement an object-oriented CAD system. We then used the techniques to implement graphical shapes, polylines, and polygons as full-fledged objects, and put them to use in our mini-CAD program. In this article, we will use the techniques we developed last time to add circles and rectangles to our program. We will also see how to add new functionality to our existing objects by implementing "copy" methods for all of them. First, let's take another look at the major features of object-oriented systems, and review how the features are implemented in our miniCAD program.

### A Second Look at Object-Oriented Programming

Objects are software models of some entity or activity. The models consist of two parts: a data part and a behavior part. The data part is a description of an object's individual data elements, and the behavior part is a description of the actions which an object performs and how it does them. The entities in CAD systems are geometric: lines, circles, text, and others. When we use a CAD program, we can move lines, resize circles, rotate polygons, and do many other things with these geometric entities. We generally think of lines, circles, and other geometric entities as passive objects which do not "do" anything. However, when we model objects using an object-oriented approach, we look at all the actions we want a program to do, and then write code which enables each type of object in the program to perform the actions of interest. We assign the actions of an object-oriented program to the software objects in the program. So, in an object-oriented CAD program, lines and circles "know" how to move themselves, resize themselves, and perform many other actions.

The data part of an object is just a conventional data structure, and the behavior part is simply a set of procedures and functions. If you look at the data structures and code we defined last time to implement shapes, lines, and polygons, you will see that they are no different from the structures and code in a CAD program which is not object-oriented. The key factor which differentiates object-oriented objects from those which are not is the linkage between the data and behavior (code) parts of the objects. An object's data is tightly coupled to its code. After an object is created, the only way to execute its code is through its data, and the only way to access or modify its data is through its code. In our object implementation, the data for an object is linked to its code with a structure pointer. The pointed-to structure contains func-

tion pointers to the object's procedures. I call the structure containing the function pointers a "class structure," since it links objects to their classes' procedures.

Object-oriented programmers create new types of objects by defining new classes. Classes are abstract (programmer-defined) types. They specify and implement the data structures and functionality for a single type of object. Classes consist of private data definitions and code for their objects, and public interfaces for applications which access and use the classes' objects. The private data structures and code for classes are their implementation. The public class interfaces specify how application code interacts with classes and their objects. The separation of a class's implementation from its specification (how to use it) is called "encapsulation." Encapsulation helps reduce a program's complexity, reduce the effort required to maintain and modify it, and increase its reliability.

In our approach to object-oriented programming, we encapsulate our classes by packaging their code and data into several files. We implement our classes with three files, and implement their application interfaces with two files. The three implementation files are a "private" include file, a "public" include file, and a source code file. Data structures and procedures used only by classes are declared in their private include file, and data structures and procedures used by application code are declared in their public include file. All the procedures which implement the functionality of a class's objects are placed in a single source code file. They are defined with C's "static" scope qualifier, which limits their visibility to their source code file.

Two files are used to specify how application code interacts with a class's objects. One file contains the source code for the procedures an application calls, and the other is just the include

file which declares the procedures. These two files define and declare a "class interface" between application code and the class's objects and their procedures. For the shape objects in our mini-CAD program, the three implementation files for their class (shape class) are shapeClass.c, shapeClass.h, and shapeClassP.h. The two class interface files for application code which uses shape objects are shape.c and shape.h. Figure 1 shows how all these files are used.

The following structure defines our shape objects:

```
typedef struct _shapeObject
{
class_p class;              /* link to class structure  */
    void    *next;          /* link to next object      */
    double  minx,miny;      /* bounding box corner      */
double  maxx,maxy;      /* bounding box corner      */
} shapeObject_t,*shapeObject_p;
```

The "class" member of the structure provides the linkage between shape objects and their code. It is a pointer to a structure named "shapeClass". This structure is defined in shapeClass.c and exported as an opaque (void*) type in shapeClass.h. The shapeClass structure contains function pointers which point to the procedures used with shape objects. Shape objects are created as shapeObject_t structures. When they are created, the "class" structure member is initialized to point to the shapeClass structure. Code which knows how shape objects are defined, and knows which function pointers are in the shapeClass structure, can call the procedures for shape objects using the shapeClass pointer. Figure 2 shows how shape objects are linked to their code with the shapeClass structure pointer.

New classes are created either from scratch, or are derived from existing classes. In Part 3 of this series, we created three classes: shape class, line class, and poly class. We created shape class as a brand new class, derived line class from shape class, and then derived poly class from line class. Classes which are derived from existing classes are called "subclasses," and the classes they are derived from are called their "superclasses." Subclass objects are like the superclass objects from which they are derived. They have the same data elements and class interface procedures as their superclass objects. Subclass objects perform the same actions as their superclass objects. They may also have data elements their superclass objects do not have, and execute actions their superclass objects do not perform.

For example, the structure defining our line objects is:

```
typedef struct _lineObject
{
    class_p class;          /* link to class structure  */
    void    *next;          /* link to next object      */
    double  minx,miny;      /* bounding box corner      */
    double  maxx,maxy;      /* bounding box corner      */
    point_p points;         /* linked list of points    */
} lineObject_t,*lineObject_p;
```

Except for the addition of a "points" member, the structure for line objects is identical to the structure for shape objects. Line class implements the same actions as shape class, but some of them are implemented differently. Line objects and shape objects do the same things when their "display()" and "pointToObject()" procedures are called, but they do them in different ways. Line class also implements actions which are not specified by shape
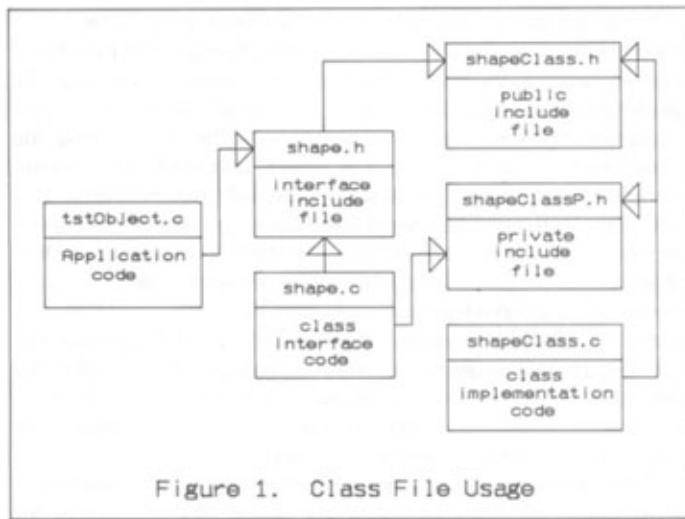


Figure 1.   Class File Usage
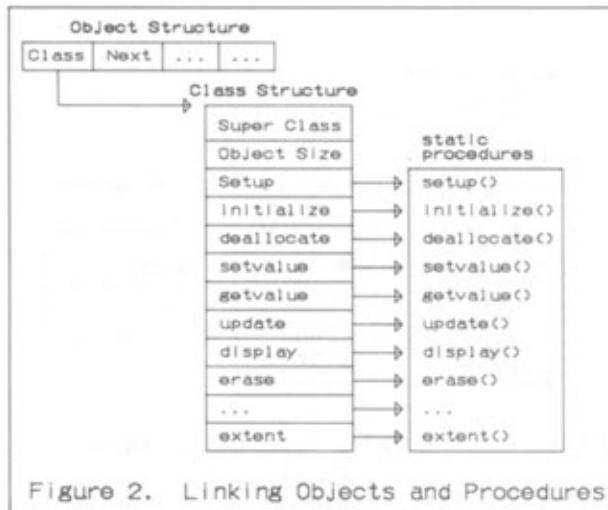


Figure 2.   Linking Objects and Procedures

class. These actions are for finding, dragging, and moving a line object's individual points. In the same way that object and subobject structures share common members, the class structures for classes and their subclasses share common members. Except for the addition of new function pointers, the class structure definition for line class is identical to the class structure definition for shape class.

Classes and their subclasses model hierarchical "is-a" relationships between objects: a line "is-a" shape, and a polygon "is-a" line. Although a line is a shape, and a polygon is a line, both are specialized versions of their superclass objects. Object systems provide "inheritance" to support modeling "is-a" relationships between objects. Subclass objects inherit both data definitions and behavior (implemented with procedures) from their superclass objects. Inheritance not only supports modeling hierarchical object relationships, it also provides a stronger form of code sharing and code reusability than that provided by code libraries. As a result, in object systems, new classes of objects can usually be created from existing classes with very little new code.

We implement inheritance in our object system with several different programming techniques. However, each technique is based on "structure overloading." A structure defined by adding new elements to the end of an existing structure "overloads" the existing structure. As an example, the structure for line objects overloads the structure for shape objects. By overloading the structure for shape objects, line objects automatically contain (inherit) the data definition for shape objects. Structure overloading also permits the procedures defined for shape objects to be used for line objects. Since the structure members common to both line objects and shape objects are in the same positions in their structures and have the same meanings for both types of objects, the shape object procedures can be safely used to manage the "shape part" of line objects. However, the opposite is not true: the line object procedures can not be used for shape objects since they access the "points" member of the line object structure, and shape objects do not have a "points" member.

The class structure for line objects, lineClass, overloads shapeClass, the class structure for shape objects. Except for
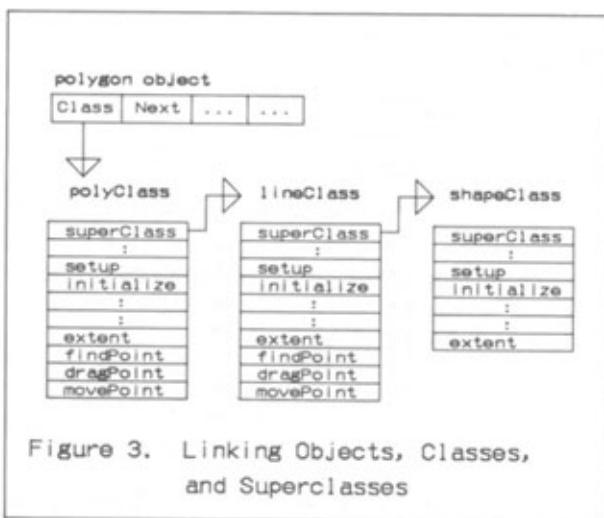


Figure 3.   Linking Objects, Classes, and Superclasses

pointers to procedures for the new actions specified by line class, the lineClass structure is identical to the shapeClass structure. It contains the same members as shapeClass, and the members are all in the same positions as they are in shapeClass. The procedures which manage and access the members of the shapeClass structure can also manage and access the members of the lineClass structure that are common to both shapeClass and lineClass. These are mainly the class interface procedures for shape class, which are in the file shape.c. Thus, by overloading the shapeClass structure, line class automatically inherits the class interface procedures defined for shape class.

What about the procedures in the file shapeClass.c which implement the actions for shape objects? They are not visible outside their file, so how can they be used with line objects? Three programming techniques allow the procedures written for shape objects to be accessed and used for line objects. The three techniques are superclass dispatching, procedure chaining, and procedure substitution. All three techniques rely on the knowledge that class structures for subclasses overload the class structures of their superclasses, and that class structures contain pointers to the class structures they overload. Figure 3 shows how objects are

linked to their class structures, and how class structures of subclasses are linked to the class structures of their superclasses.

Superclass dispatching is the simplest of the techniques. Subclasses know which procedures are in their superclass, know how to access the class structure pointer for their superclass, and know how to access pointers to their superclass's procedures from the class structure. Thus, procedures in subclasses can call procedures defined by their superclasses. Superclass dispatching occurs when a procedure in a subclass accesses and uses a function pointer to call (dispatch to) a corresponding procedure in its superclass. Superclass dispatching is commonly used to augment the actions performed by a subclass's objects. Line class's setvalue() and getvalue() procedures perform superclass dispatching by calling shape class's setvalue() and getvalue() procedures.

Procedure chaining is easier to explain with an example. Object destruction is a two-step process: first, any internally-allocated memory is freed by calling the "deallocate" procedures of an object's class and all its superclasses, then the memory allocated for the object itself is freed. Here is a code fragment from deleteShape() showing how objects are freed:

```
int deleteShape( object_p *object )
{
    class_p         class = getObjectClass(*object);
    shapeClass_p    shapeC = getShapeClass(class);

    while ( shapeC )
    {
        if ( shapeC->deallocate )
            (void)(*shapeC->deallocate)(*object);
        shapeC = getShapeClass(shapeC->superClass);
    }
    ...
}
```

The "while" loop follows the chain of superclass structure pointers stored in the class structures. Inside the loop, the "deallocate" procedures for an object's class and all its superclasses are called to free any internally allocated memory. The process of calling the "deallocate" procedures for the object's class and all its superclasses by following class structure pointers is called procedure chaining. If the procedures are called beginning with an object's class, then proceeding up through its superclasses, the process is called "upward chaining." If the procedures are called starting with shape class, then proceeding down through subclasses until the object's class is reached, the process is called "downward chaining." Upward chaining is implemented by iterating through class structures, following the superclass pointers stored in them, and calling procedures for each class. Downward chaining is implemented with recursion: recursive procedure calls are made for each superclass until the root class (shape class) is reached. As the recursive procedure calls unwind, the procedures for each of an object's superclasses are called. Downward chaining is used in createShape() to initialize data values for newly created objects, and upward chaining is used in deleteShape(). Both of these procedures are in the source file shape.c. Figure 4 shows the order in which procedures are called when procedure chaining is used.

The third programming technique we use to implement inheritance is procedure substitution. This occurs when a super-class substitutes a pointer to a "real" procedure for a "dummy"

procedure pointer in the class structure of one of its subclasses. A class specifying a procedure defines a dummy inheritance procedure corresponding to the real procedure and makes it available to its subclasses. Subclasses that need to inherit the real procedure place the pointer to the dummy procedure in their class structures. The first time an object is created, its class structure is initialized using superclass dispatching: the class structure is sent to each of the setup() procedures of the class's superclasses. The setup() procedure in each class defining a procedure examines the class structure sent to it, looking for a pointer to a dummy inheritance procedure corresponding to a real procedure it defined. If the pointer to a dummy procedure is found, it is replaced by a pointer to a real procedure defined by the class. Procedure substitution is implemented in the setup() procedure of each class. You can see how procedure substitution works by tracing through the class setup() procedures to see how poly class inherits the "move" procedure defined by line class, and inherits the "moveDrag" procedure defined by shape class.

Structure overloading also lets us implement "polymorphic procedures" in our object system. Polymorphic procedures can be called for any object, regardless of their type. The polymorphic procedures take care of selecting and calling the procedures which are correct for the type of object sent to them. In our object system, polymorphism is implemented using overloaded object structures, overloaded class structures, and function pointers. Since all our objects are created by overloading the shape object structure, and all our class structures are created by overloading shape class's class structure, the class interface procedures defined for shape class can be used for all our classes and objects. To display any object, no matter what type of object it is, our application code calls displayShape(). The same is true for all the class interface procedures in shape.c. They are all polymorphic procedures. Because our objects and class structures are defined with structure overloading, the class interface procedures for a class are polymorphic for it and all of its subclasses. However, the opposite is not true. For example, the procedures defined in line.c can be used for lines and polygons, but not for shapes, since shape objects do not have a "points" structure member, and the shapeClass structure does not have the findPoint(), movePoint(), and dragPoint() function pointers which are in the lineClass structure. Because of this, the polymorphic procedures in line.c need to make sure the objects sent to them are not shape objects. They make this determination by calling isSubClass(). This procedure iterates through an object's class structures (using the "superClass" pointers) looking for a class structure pointer that matches the line class structure pointer. If a match is found, the object is either a line object or a polygon object. In this case, the object's class structure contains the function pointers the polymorphic line procedures access and call. Notice that isSubClass() does not perform actual low-level type checking on an object: it only determines if an object is a member of a general class of objects. Figure 5 shows how the polymorphic procedures for shape class and line class select procedures by accessing class structures.

Now that we have broadly reviewed the object system we implemented last time, we will now see how to extend it to include new functionality and new classes of objects.

## Adding New Functionality to Existing Classes

A common CAD function we have not implemented is a "copy" function. Not only is this a common function, it is a mandatory function for any self-respecting CAD system. So we will now add a copy function to our mini-CAD program, and here is how it will work:

To copy an object, a user will select "copy" from the "action" menu. Next, she or he will move the mouse cursor over the object to be copied and select the object by pressing mouse button 1. The picked object will be copied and the copy will be highlighted. The user will move the highlighted copy to a new location by dragging it with the mouse. When the copy is positioned, the user will place it at its new position by pressing mouse button 1. The copy will be unhighlighted and displayed at its new location. The copy action can be aborted by leaving the copy at its original location. In this case, the copy will be unhighlighted and deleted. The copy action will remain in effect until a new menu action is picked.
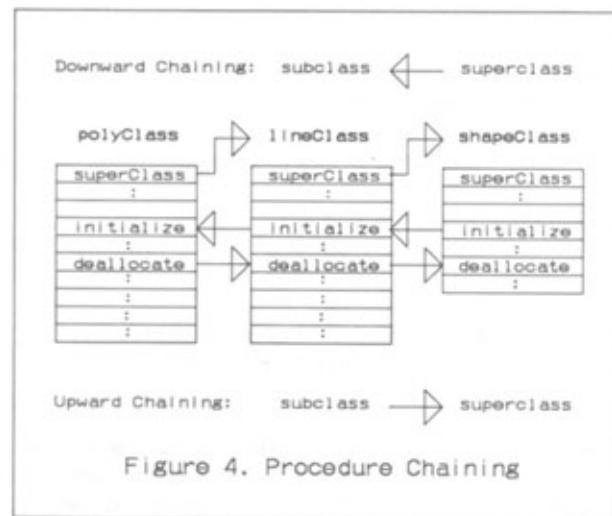


Figure 4. Procedure Chaining

We need to do several things to implement the copy action. We need to add a copy menu item to our action menu and write a copy "action" procedure to orchestrate the actions specified above. We also need to link the copy menu item to its action procedure. More importantly, we need to add copy procedures to our classes, and add a copy class interface procedure for our action procedure to call.

Our copy action procedure is named pickAndCopy(). We link the copy menu item to our copy action procedure the same way we linked our menu items to their code last time: by overloading Intuition's menuItem structure and adding an "intuiExtension_t" structure to it. The overloaded menuItem structure is called "myMenuItem_t". It is defined in "globalDefs.h". The intuiExtension_t structure contains a pointer to an event handler procedure that is called when a menu item is picked. It also contains a pointer to a data structure called "miData_t". The miData_t structure contains a pointer to our copy action procedure, pickAndCopy(). When the copy menu item is selected, our event input handler, handleInput(), retrieves the event handler function pointer from the overloaded menuItem

structure pointer. HandleInput() then calls the event handler, sending it the miData_t structure pointer. The copy menu item event handler is "miSetAction()". It retrieves the pointer to our pickAndCopy() action procedure from the miData_t structure pointer and places it into our program's global state vector, which is called "world". Later, when a button press occurs, handleInput() calls "windowEvent()" to take care of it. If an action procedure pointer is in the world state vector, windowEvent() calls it to process the button press event. For our copy menu item, pickAndCopy() is the action procedure windowEvent() calls when a button press occurs. The copy menu item is defined and initialized in the file menu.c, and pickAndCopy() is defined in tstObject.c and declared in ieHandlers.h. You may want to review how we linked Intuition's input objects to our application code and managed input events. The way this is done is explained in excruciating detail in Part 2 and Part 3 of this series.

The action procedure to copy an object is almost identical to the action procedure to move an object. The only difference is that instead of moving whatever object is picked, we first copy the picked object, move the copy, then add it to the list of world objects. Here is the algorithm to copy an object:

> If an object is not picked, return.
> Copy the picked object.
> Highlight the picked object.
> Process "move drag" to obtain delta coordinate values for positioning the copy.
> Unhighlight the picked object.
> If the delta coordinate values are zero (copy not moved), delete the copy.
> Otherwise,
> > Insert the copy in the list of world objects.
> > Move the copy to its specified location.
> > Display the copy.

As the algorithm shows, the action procedure to copy an object is simple. Except for procedures to actually copy objects, we already have all the procedures to implement our copy action procedure using this algorithm. Since all our objects are implemented with object-oriented programming techniques, we need to provide our classes with the ability to copy their objects. Each class either needs a copy procedure added to it, or needs to inherit a copy procedure from one of its superclasses. We will also need a polymorphic class interface procedure our application code can call to invoke an object's copy procedure.

The mechanics of adding a new procedure to an existing class are relatively simple: a function pointer for the new procedure is added to the class's structure definition, the procedure is defined, a dummy "inherit" procedure is defined, and finally, a class interface procedure is written. However, when a class structure definition is modified, the class structure definitions of all its subclasses are affected. This is because they overload the class structure definition of their superclass. Before adding new functionality to a class, we need to identify which class the functionality will be added to and find all its subclasses so we can update them. Since we need the ability to copy any object, we will add the copy protocol (the procedure, its linkage, the way it is expected to work, and its class interface procedure) to shape class, the root class of our class hierarchy. We will need

to update line class and poly class since they are subclasses of shape class.

What needs to be done when an object is copied? First, space for the copy needs to be allocated, then the values from the original object need to be copied into the new object. If the original object does not contain pointers to any dynamically allocated memory, the copy is complete. However, if the original object contains pointers to dynamic memory, the dynamic memory also needs to be copied and a pointer to it needs to be placed into the new object. How does our mini-CAD program know which objects contain pointers to dynamic memory, and how does it make sure the pointed-to data gets copied? It doesn't! The only place this type of detailed knowledge about objects is available is in their classes.

Our object structures are formed by combining structure "parts." Each subclass can define a new "part" and add it to an existing object structure. If an object part defined by a class contains a pointer to dynamically allocated memory, that class needs to be called when an object containing the part it defines is copied. As an example, the object structure for a polygon is formed by adding a "line part" to a "shape part." The line part is a pointer to a linked list of points. When a polygon is copied, the linked list of points needs to be copied. Since line class defines the "line part" of lines, which is inherited by polygons, line class is responsible for copying the points. The bottom line is this: when an object is copied, each class which contributes dynamically allocated members to the overall object structure needs to be called to take care of copying the dynamic memory. If a class does not define any dynamic members for the object structure, the class does not need to be called when the object is copied. To make sure objects are copied correctly, our copy class interface procedure will chain the objects' copy procedures in superclass to subclass order.

Keeping the above discussion in mind, here is a step-by-step description of how copy functionality is added to our classes:

Step 1: Identify the class which specifies new functionality and add a function pointer to the class's structure part.

Add the copy functionality to shape class so that all objects can be copied. Add a "copy" function pointer to the SHAPEC_PART macro defined by shape class in the private include file, shapeClassP.h. The copy function pointer is "int (*copy)(object_p object,object_p copy)".

Step 2: Identify all subclasses of the class modified in step 1, and add the new procedure to the class and each of its subclasses. Make sure the class structures contain the new function pointer structure member.

The class structures for all three of our classes (shape class, line class, and poly class) need to be updated. Since line class defines a dynamically-allocated member for line objects, line class needs a copy procedure. Shape class does not need a copy procedure, and neither does poly class, since line class takes care of copying points for polygon objects. Set the "copy" function pointers for the shapeClass structure and the polyClass structure to NULL. These two structures are initialized in shapeClass.c and polyClass.c. Define a copy procedure for line class, and set the copy function pointer in the lineClass structure to point to the procedure.

**Step 3**: Define a dummy "inherit" procedure and a class interface procedure for the new function.

Since shape class specifies the copy functionality, it is responsible for defining the "inherit" procedure and the class interface procedure. The class copy procedures are downward chained (called in superclass to subclass order), so they can not be inherited by subclasses. Thus, for the copy procedure, we do not need to define an "inherit" procedure.

From the above steps, we see that to add copy functionality to our classes, we only need to write two new procedures: a copy class interface procedure for shapes, and a copy procedure for line class. Here is line class's copy procedure:

```
static int copy( object_p object,object_p copy )
{
    lineObject_p line = (lineObject_p)object;
    lineObject_p cpy  = (lineObject_p)copy;
    point_p       point,new,last;

    cpy->points = last = NULL;
    point = line->points;

    while ( point )
    {
      new = (point_p)malloc(sizeof(point_t));
      if ( ! new )
      {
        printf(
        "lineClass.copy:  can't allocate memory.");
        return 1;
      }
      new->x = point->x;
      new->y = point->y;
      new->next = NULL;
      if ( last )
         last->next = new;
      else
         cpy->points = new;

      last = new;
      point = point->next;
    }
    return 0;
}
```

Line class's copy procedure initializes the "points" member in the copy of the original object to NULL, then loops through the list of points in the original object, copying and linking them together. This procedure is added to lineClass.c.

The new class interface procedure for accessing the classes' copy functionality is added to shape.c. Its prototype is added to shape.h. Here is the procedure:

```
object_p copyShape( object_p object )
{
    shapeObject_p  shape = (shapeObject_p)object;
    shapeClass_p   shapeC = (shapeClass_p)shape->class;
    object_p       copy = NULL;

    if ( ! shape || ! shapeC )
      return( NULL );

    /*
     * allocate memory for the copy and copy values from
     * 'shape' into 'copy'
     */

    copy = (shapeObject_p)malloc(shapeC->objectSize);
    if ( ! copy )
```
```
    {
      printf("copyShape:\n");
      printf("  memory allocation failed.\n");
      return NULL;
    }

    memcpy((void*)copy,(void*)object,shapeC->objectSize);

    /*
     * chain down through classes to complete copy action
     */

    if ( copyObject( object,(object_p)copy,shapeC ) )
    {
      (void)deleteShape( &copy );
      return NULL;
    }
    return copy;
}
```

The procedure allocates memory using the number of bytes required for an object (stored in the class structure) and performs a bytewise copy of the original object into the new space. At this point, if the object does not contain any dynamically allocated memory, the copy is complete. However, copyShape() does not know this, so it calls copyObject() to chain through all the class structures for the class and its superclasses and call any copy procedures it finds in the class structures. Here is copyObject():

```
static int copyObject( object_p object,object_p copy,
                       class_p class )
{
    class_p super = getSuperClass(class);
    shapeClass_p  shapeC = getShapeClass(class);
    int           tmpRet = 0,retVal = 0;

    /*
     * recursively call self until we get to the root
     * class (has no superclass).  Then, as recursion
     * unwinds, call each class which has a copy method.
     * Make sure all superclasses are called even if
     * one fails - otherwise, if we delete the object
     * because of an error return, a pointer in the
     * original object may be unintentionally freed.
     */

    if ( super )
      retVal = copyObject(object,copy,super);

    /* call each class as we return from recursion */
    if ( shapeC->copy )
      tmpRet = (*shapeC->copy)(object,copy);

    return tmpRet > retVal ? tmpRet : retVal;
}
```

Other than being recursive, which is always confusing, copyObject() is pretty simple. Notice that even if a class copy procedure returns an error, the recursion continues until all the object's copy procedures are called. If copyShape() fails, the copy of the object will be deleted by calling all its classes' "deallocate()" procedures, so each class needs to make sure any copied pointers are initialized to a known state.

To see how the copy procedures work, let's step through what happens when a polygon is copied. First, we select "copy" from the "action" menu. Our event handler calls miSetAction(), which places a pointer to pickAndCopy() in the program's state vector, "world". Next, we move the cursor over a polygon and select it. The event handler calls windowEvent() to take care of the button press. WindowEvent() looks for a pointer to an action

procedure in the state vector, finds the pointer to pickAndCopy(), and calls it. PickAndCopy() calls findObject() to see if an object was picked. FindObject() returns the pointer to the polygon we picked. PickAndCopy() then calls copyShape(), our new class interface procedure. CopyShape() allocates memory for the polygon copy, and copies the values from the original polygon into the allocated memory. It then calls copyObject(). CopyObject() recurses, following superclass pointers until it reaches the shapeClass structure, which does not have a superclass. At this point, copyObject() starts looking for pointers to copy procedures in the class structures. The shapeClass structure does not have a copy procedure, so copyObject() returns, popping the lineClass structure pointer off the stack. The lineClass structure pointer does have a copy procedure, so it is called. It copies the list of points from the original polygon into the copy and returns. Once again, copyObject() returns. This time it pops the polyClass structure pointer off the stack. The polyClass structure pointer does not have a copy procedure pointer, so copyObject() again returns, completing the recursive calls and returning to copyShape(). CopyShape() then returns the pointer to the newly created copy to pickAndCopy(). PickAndCopy() completes the copy action by calling moveDrag() to obtain a location for the copy, inserting the copy in the list of world objects, moving it, and displaying it.

After we add these procedures to shape.c and lineClass.c, modify the class structure macro in shapeClassP.h, modify the class structures for all our classes, and recompile and link our tstObject program, we can create copies of any object. The procedures will even work without any modifications when we add new classes of objects to our program, which we will now do.

## Adding New Classes of Objects

Extensibility is one of the major benefits of object-oriented systems. It is usually very easy to add new classes of objects to a base set of existing classes. If the existing classes are designed and implemented so their procedures can be inherited by new classes, many new classes can even be added with very little code. Companies and software vendors are cashing in on this aspect of object-oriented systems by creating and marketing object-oriented application development frameworks. These object-oriented frameworks are just a base set of classes along with tools and documentation for creating new classes as subclasses of the base classes. The classes we developed last time, along with the techniques for adding new classes, constitute a stripped-down, manual framework for developing object-oriented two-dimensional CAD applications.

To add new classes to our (or any) framework, we need to know several things. First, we need to know what new classes to add. This is usually pretty simple to figure out. Next, we need to know which classes already exist, know how they are organized in an inheritance hierarchy, and understand the protocol they specify. The protocol consists of the procedures classes are expected to provide or inherit, how the procedures are called, and what the procedures are expected to do when they are called. We need to understand the protocol so we can determine which procedures to write and how to write them, and we need to know the existing class hierarchy so we can decide where to add new classes in the hierarchy. Finally, we need to know the mechanical details of how to "bind" new classes to a framework.

A minimum set of objects for professional CAD systems includes lines, polygons, arcs, circles, rectangles, text, and splines. Most CAD systems offer more objects than these, but some do not. Our mini-CAD system already has lines and polygons, so we need to add arcs, circles, rectangles, text, and splines to it. Of these five candidate classes of objects, we will add classes for circle and rectangle objects. If you understand how these two classes are implemented and added to our framework, you should be able to add the remaining classes of objects to the program. The low-level code for interactively creating and editing arc, text, and spline objects is pretty complex, but the techniques for adding the classes to our framework are identical to those for adding circles and rectangles.

New classes are added to object frameworks as subclasses of existing classes. This is called "subclassing" by object-

## TABLE 1. Shape Class Methods

| Method Required | Chained | Inheritance | Symbol |
| --- | --- | --- | --- |
| setupClass | yes | upward | none |
| initialize | no | downward | none |
| deallocate | no | upward | none |
| copy | no | downward | none |
| setvalue | yes | no | none |
| getvalue | yes | no | none |
| update | yes | no | none |
| display | no | no | inheritDisplay |
| erase | no | no | inheritErase |
| highlight | no | no | inheritHighlight |
| unhighlight | no | no | inheritUnhighlight |
| insertdrag | no | no | inheritInsertdrag |
| movedrag | no | no | inheritMovedrag |
| sizedrag | no | no | inheritSizedrag |
| rotatedrag | no | no | inheritRotatedrag |
| move | no | no | inheritMove |
| resize | no | no | inheritResize |
| rotate | no | no | inheritRotate |
| pointToObject | no | no | inheritPointToObject |
| extent | no | no | inheritExtent |

oriented programmers. Since subclasses inherit data and procedures from their superclasses, deciding which existing class to subclass has a significant effect on how much work is needed to add a new class, and on how well it will work after it is added. Subclass objects are specialized versions of their superclass's objects, so a good way to decide which class to subclass is to find one whose objects are similar to the new objects. As an example, polygons are almost identical to polylines, so we added polygons to our class hierarchy by subclassing line class. Let's look at our class hierarchy to see where to add rectangle class and circle class.

Our existing class inheritance hierarchy is shown in Figure 6. Shape class is at the top of the hierarchy. Shape class models the behavior of a broad class of CAD objects, and the shape object data structure (a bounding box) is a general representation for graphical objects. All classes are subclasses of shape class. Line class extends the shape class model to include graphical objects which can be modeled as sequences of points connected by line segments. Poly class models polygons by specializing polylines, forcing their first and last points to coincide.

Rectangles are almost identical to shapes. They can be defined with the same data structure, and shape class already specifies the behavior we need for rectangles. For these reasons, we may be tempted to create rectangles by subclassing shape class. However, there is one catch: the shape object data structure models the horizontal and vertical positions and dimensions of any graphical object, so shape objects can only be rotated in 90-degree increments. If we create rectangle class by subclassing shape class, we will have to override shape class's "rotate"

procedure so our rectangles can be rotated at arbitrary angles. To rotate rectangles at arbitrary angles, we need points representing their corners in addition to the data describing the bounding boxes around them. But we already have classes of objects whose data structures consist of bounding boxes and points, and whose objects can be rotated at any angle. They are line class and poly class. Rectangles are closed polygons. However, they are specialized polygons: they always consist of four sides, and the sides are joined at right angles. Since rectangles are specialized versions of polygons, we will create rectangle class by subclassing poly class, overriding and specializing poly class's "insertDrag" and "resize" procedures.

Which class do we subclass to create circles? We could use polygons as approximations of circles, and subclass poly class. However, calculating and drawing the edges needed to approximate a circle can be time consuming, especially when they are being resized or moved around. Besides, it is more "natural" to model circles as objects with center points and radii. This model lets us easily calculate diameters, circumferences, and areas of circle objects. Our existing class hierarchy does not contain objects which are similar to circles, so we will create circle class by subclassing shape class.

In case you want to try your hand at adding arcs, splines, and text objects to our framework, let's briefly consider which classes to subclass to create these three classes of objects. Arcs are specialized versions of circles. Actually, circles can be modeled as specialized arcs, but since we are adding circles instead of arcs, arcs can be added by subclassing circles. Splines are similar to polylines and polygons. Splines, like polylines and polygons, consist of points connected by edges. The edges for polylines and polygons are simple line segments. For splines, the edges are curves whose curvature is determined by the locations of the splines' points. Open splines can be added to our class hierarchy by subclassing line class, and closed splines can be added by subclassing poly class. Our class hierarchy does not have any class of objects similar to text objects, so text objects can be added to the hierarchy by subclassing shape class (text class will need a drag input handler which interacts with keyboard events instead of mouse motion events). Figure 7 shows the entire class hierarchy for implementing a complete set of CAD objects with our framework.

## TABLE 2. Shape Class Method Actions

| Method | Action |
|---|---|
| setupClass | Initialize class structure. Resolve inheritance for subclass structures. |
| initialize | Initialize object structure part, especially internal pointers. |
| deallocate | Free internally allocated memory when object is destroyed. |
| copy | Copy internally allocated memory when object is copied. |
| setvalue | Modify object data values. |
| getvalue | Return object data values. |
| update | Validate object data value modifications after setvalue is complete. Allow, disallow, or change modifications. Perform additional actions if needed. |
| display | Graphically display self. |
| erase | Graphically erase self. |
| highlight | Graphically highlight self. |
| unhighlight | Graphically unhighlight self. |
| insertdrag | Process graphical interaction to establish initial object values when object created. |
| movedrag | Process graphical interaction to obtain delta values to move self. |
| sizedrag | Process graphical interaction to obtain delta values to resize self. |
| rotatedrag | Process graphical interaction to obtain delta value to rotate self. |
| move | Move self by delta coordinate values. |
| resize | Resize self by delta scale values. |
| rotate | Rotate self by delta angle value |
| pointToObject | Return distance from a point to self. |
| extent | Return bounding box coordinates. |

Now that we have decided which objects to add to our framework, and decided where to add them in the class hierarchy, the next step is to look at the class protocol to determine which procedures our new classes need. A complete description of a class's protocol consists of two "specifications", and can be quite voluminous. One part of the protocol specifies how application code interacts with a class's objects. The other half of the protocol specifies the procedures classes are expected to provide, and specifies how they are expected to work within the class hierarchy. The function prototypes in the include files for our class interface procedures (shape.h and line.h) are part of the protocol for application-object interaction. The function prototypes in the private include files for our classes are part of the second half of the protocol. However, in addition to information in the private include files, we also need a description of which procedures are required, which can be inherited, which are chained, and which ones can be overridden. A simple way of presenting this type of information is in a table. Table 1 shows this information for shape class, and Table 2 describes the action each of the methods is expected to perform. Tables 3 and 4 present the same information for the methods specified by line class.

We see from the above tables that the only methods which can *not* be inherited by subclasses are those which are required or are chained. Since shape class specifies the method protocol for all our objects, it provides default procedures for each of the methods it specifies. It also provides dummy "inherit" procedures for each of its methods that can be inherited. Line class specifies the method protocol for itself and any of its subclasses, so it also provides default methods and dummy "inherit" procedures for the methods it specifies. Poly class does not specify any new methods, so its protocol is the same as line class's protocol.

## TABLE 3. Line Class Methods

| Method | Required | Chained | Inheritance Symbol |
|---|---|---|---|
| findPoint | no | no | inheritFindPoint |
| dragPoint | no | no | inheritDragPoint |
| movePoint | no | no | inheritMovePoint |

We now know which procedures our new classes are required to provide and which ones can be inherited, and have an idea about what our classes' procedures are supposed to do. We are almost ready to implement them. Before actually implementing a new class, we need to answer several questions. These questions are relevant in any object system. However, the answers to the questions depend on how new classes are added to the particular object framework, or are defined using an object-oriented language. The questions, and the answers for our framework, are:

1. Will the new class need to specify protocol in addition to the protocol specified by its superclass? That is, will the new class require methods in addition to those of its superclass?

If the answer to this question is "yes", then the new class will need to define a "class part" containing pointers to the new methods. The class part is appended to the class structure of a new class's superclass, overloading it. It will also need to define dummy inheritance procedures for any future subclasses, and will need to provide class interface procedures so application code can access its new methods. Line class is an example of a class which extends the protocol specified by its superclass.

Neither of our new classes needs methods other than those specified by their superclasses. Nevertheless, we will define an empty "class part" (using a macro) for our new classes. We do this for two reasons. The first reason is for consistency in defining classes. The second reason is to make adding methods at some later date easier.

2. Will the new class's objects need data elements in addition to the data elements contained in their superclass's objects?

If the answer to this question is "yes" (it usually is), the new class needs to define an "object part" containing the new data elements for its objects. The object part is appended to the object structure of a new class's superclass, overloading it. The class will also need to define "atoms" for the new data elements so application code can access and modify the data elements using our getShapeValues() and setShapeValues() interface procedures.

Circle class needs to define new data elements to describe circles, but rectangle class does not need to extend the object structure defined for lines and polygons. We will define an empty "object part" for rectangles anyway.

3. Could the new class be useful as a superclass for another class of objects at some future time?

Unless a new class is rather specialized, the answer to this question is usually "yes." If so, the class should be designed and implemented so that it can be subclassed. Sometimes this is not easy to do.

Both circle class and rectangle class are good candidates for subclassing. Circle class can be used to derive an "arc class," and rectangle class can be subclassed to create a "rounded rectangle" class. We will implement our new classes so they can be subclassed.

### Implementing Circle Class

The first task to tackle when implementing a new class is to define the data structures for the class and its objects. In our framework, data structures are defined in the class's private include file. For circle class, the private include file is circleClassP.h (shown in Listing 1). The structures we need to define are the class structure and the object structure for circle class and its objects. Circle class does not need to extend the protocol specified by shape class, so we do not need to define class part data elements for its class structure. We simply define an empty macro for the new class part, and append the macro to the class parts defined by

shape class, creating a class structure typedef'ed as circleClass_t. To manage circle objects, we need to know their center points and radii. We could actually extract this information from the bounding box defined for shape objects, and inherited by circle objects. However, the bounding boxes for objects should be calculated from the dimensions of the objects, rather than the other way around. Thus, the structure for circle objects needs to contain their center x,y coordinates and their radius. We define a circle part macro, consisting of these data elements, and append it to the object parts defined by shape class to create a circleObject_t structure.

Private include files are included in their class's source code file, and also included by subclasses. In addition to data structures, they also contain declarations of any variables and procedures which subclasses may need or find useful. Subclasses of circle class will need access to its class structure so they can use it for their superclass; thus the class structure _circleClass is declared in circleClassP.h. Circle class defines several procedures which may be useful to subclasses, so they are also declared in circleClassP.h.

After the private include file for a class is complete, we create its public include file. This file is included by both the class's source code file and by application code. The public include file exports a pointer to the class's class structure, and exports "atoms" which identify data elements of the class's objects. Application code uses the class structure pointer to create the class's objects, and uses the atoms to query and modify an object's data values. The public include file for circle class is in Listing 2.

The final steps in creating a class (which does not extend the protocol specified by its superclass) are to define and initialize the class's atoms and its class structure, and to write the methods for the class. This is done in the class's source code (implementation) file. The implementation file for circle class is presented in Listing 3. Before we can define and initialize the class structure, we need to decide which methods to write and which ones to inherit, since pointers to the methods and the dummy inheritance procedures are placed in the class structure.

Looking at Tables 1 and 2, we see that new classes are required to provide setupClass, setvalue, getvalue, and update methods. New classes can not inherit initialize, deallocate, or copy methods, since these are chained methods. Circle class does not define any dynamic data elements (pointers to internally allocated memory) for circle objects, so circle class does not need to provide initialize, deallocate, or copy methods. The remaining methods specified by shape class can be inherited or defined by subclasses. All these methods involve the geometry of objects. Since circles and shapes are not geometrically very similar, circle class needs to define most of these methods. The only ones which can be sensibly inherited are movedrag, sizedrag, rotatedrag, and extent. Although the three drag methods can be easily implemented (and probably should be), circle class will inherit them.

The source code files for our classes are organized in four logical sections: the first section contains forward declarations of the classes' procedures, the second section defines and initializes the classes' atoms, the third section defines and initializes the classes' class structure, and the fourth section contains definitions of the classes' methods. The first three sections are fairly self-explanatory, so we will concentrate on the implementation of circle class's methods.

Automatic method inheritance is implemented by our classes' setup() procedures. Circle class's setup() procedure is called the first time our program calls createShape() to create a circle or an object in one of circle class's subclasses (currently none). If the circleClass structure has not been initialized, setup() calls its superclass's setup() procedure, sending it the pointer to the circleClass structure. Superclasses are responsible for resolving inheritance for their subclasses, so shape class replaces the pointers to the "inherit" procedures in the circleClass structure with pointers to the actual procedures defined by shape class. Circle class's setup() procedure then examines the class structure pointer sent to it. If the structure pointer does not point to the circleClass structure, it is a pointer to a class structure of one of circle class's subclasses. If this is the case, circle class is responsible for replacing any "inherit" procedure pointers with pointers to procedures it defines corresponding to the dummy procedures. The setup() procedure does this by comparing function pointers in the class structure with pointers to the dummy procedures. If the pointers match, the dummy procedure pointers are replaced with pointers to procedures defined by circle class. The setup() procedure then sends the class structure pointer on up to shape class's setup() procedure to finish initializing it.

Circle class's getvalue() and setvalue() procedures are called by getShapeValues() and setShapeValues(), which are called from our program's pickAndEdit() and updateValues() procedures. Both the getvalue() and setvalue() procedures examine the "arg" array sent to them, looking for the atoms defined by circle class. When one of circle class's atoms is found, getvalue() assigns the appropriate value from the circle object to the "argPtr" in the "arg" record. Likewise, setvalue() assigns the value from the "argPtr" in the "arg" record to the circle object. These assignments are made using typecasts so the values will be assigned

## TABLE 4. Line Class Method Actions

| Method | Action |
| --- | --- |
| findPoint | Return pointer to the object's point which is within tolerance of an input point. |
| dragPoint | Process graphical interaction to obtain delta values to move object's point. |
| movePoint | Move object's point by delta coordinate values. |

correctly. When either of these procedures encounters an atom it does not recognize, it calls its superclass's getvalue or setvalue method to take care of the atom.

As setvalue() changes values for circle objects, it updates any of their other values related to the changed values. However, circle class's setvalue() procedure has no way of knowing when the coordinates of a circle object's bounding box are changed by shape class's setvalue() procedure. None of our class's setvalue methods know when a setvalue method in a superclass changes an object's values. So that classes can maintain the internal consistency of their objects' data values, setShapeValues() calls their update methods after an object's values have been modified.

Before setShapeValues() calls an object's setvalue() procedure, it makes a "shallow" copy of the unmodified object. After the setvalue() procedure returns, the object's update method is called. Both the modified object and the unmodified copy are sent to the update() procedure. The update() procedure compares values in the modified object with values in the copy of the unmodified object to determine which values were changed by the setvalue() procedures.

Circle class's setvalue() procedure updates the bounding boxes for circles when their radii or center coordinates are changed. Thus, the data for circle objects are kept consistent when their radii or centers are changed. However, circle class's setvalue() procedure is not responsible for the bounding box atoms: it sends



```
Procedures in shape.c
are polymorphic for
shape class part procedures

polyClass        lineClass        shapeClass

superClass       superClass       superClass
   :                :                :
initialize       initialize       initialize
   :                :                :
extent           extent           extent
findPoint        findPoint
dragPoint        dragPoint
movePoint        movePoint

Procedures in line.c
are polymorphic for
line class part procedures
```

Figure 5. Subclasses and Polymorphic Procedures

these atoms to its superclass's setvalue() procedure. Because of this, the bounding box coordinates of circles are changed by shape class's setvalue() procedure. When this happens, the bounding box values will not be consistent with the radii and center values. To take care of this condition, circle class's update() procedure compares the unmodified circle's bounding box coordinates with the modified circle's bounding box coordinates. If the bounding box values differ, the circle object was modified, and the update() procedure adjusts its values so the bounding box values, the radius, and the circle's center are consistent. Circle class's update() procedure also makes sure the bounding boxes for circles are not modified to have unequal widths and heights.

Circle class's display(), erase(), highlight(), and unhighlight() procedures are almost identical. They call SetAPen() to set the graphics foreground drawing pen and then call drawCircle(). DrawCircle() calls the graphics library procedure DrawEllipse() to draw circles. DrawEllipse() requires two radii: a horizontal radius, and a vertical radius. In world coordinates, these two radii may be different for ellipses, but they are the same for circles. Unless the pixel x to y aspect ratio of the screen is 1 to 1, the radii in screen coordinates for circles will be unequal. To account for the screen aspect ratio, drawCircle calls worldDistToViewDist() and sends it the circle's world coordinate radius as both the horizontal distance and vertical distance. WorldDistToViewDist() transforms the radius values into horizontal and vertical view coordinate (pixel) distances, which are then used in the call to DrawEllipse().

The procedure called to obtain the initial position and size of a newly created circle object is insertdrag(). This procedure sets up an array describing the new circle in view coordinates, then calls handleDrag() to process the actual drag interaction. The array describing the circle is named 'cp'. Its data consists of the circle's center x- and center y-coordinates, and the circle's horizontal and vertical radii. The pick coordinates sent to insertdrag() are used as the circle's center coordinates, and its radii are initially set to 0. The "drag draw" procedure handleDrag() calls to erase and redraw the circle as the drag interaction progresses is called dragInsertCircle(). This procedure erases and redraws the circle by calling DrawEllipse(). When handleDrag() calls dragInsertCircle(), it sends it the change in the position of the mouse cursor from the initial pick point to the current cursor location as x- and y-pixel offsets. DragInsertCircle() has to find the radius of the circle centered at the initial pick coordinates whose circumference also passes through the x- and y-offset coordinates. It also has to account for the x to y aspect ratio of the screen. To account for the aspect ratio, dragInsertCircle() calls viewDistToWorldDist() to convert the pixel offset distances to world coordinate distances. It then uses the distance formula to find the world coordinate distance from the circle center to the cursor location (refer to Figure 8). This distance is the radius of the circle. Finally, worldDistToViewDist() is called to determine the horizontal and vertical pixel distances corresponding to the world coordinate radius, and these values are used to draw the circle by calling DrawEllipse(). After handleDrag() returns to insertdrag(), the same technique is used to calculate the circle's radius from horizontal and vertical offsets. After the radius is calculated, insertdrag() initializes the circle object's data values and returns.

Circle class inherits shape class's movedrag(), sizedrag(), and rotatedrag() procedures. These procedures are called to



Figure 6. Existing Class Hierarchy
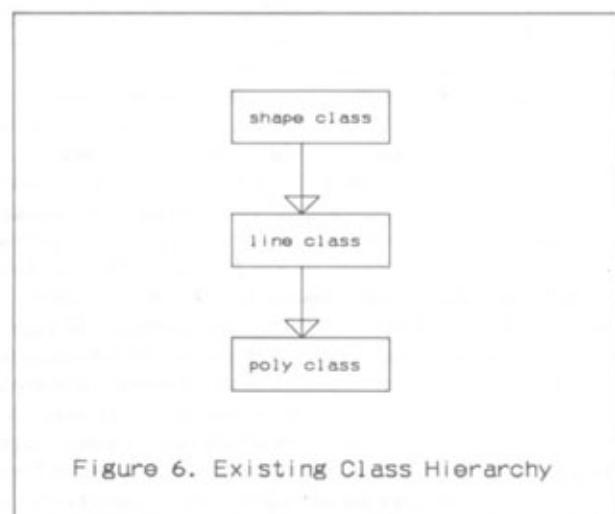
interactively obtain delta coordinate values to move an object, relative scale values to resize an object, and a delta angle value to rotate an object. The circle class procedures which perform the move, resize, and rotate actions are move(), resize(), and rotate(). Circle objects are moved by simply adding delta coordinate values to their bounding box coordinates and their center coordinates. Shape class's sizedrag() procedure does not constrain

objects to resize with equal x- and y-scale values: objects can be "stretched". If circle class's resize() procedure simply applied the x- and y-scale values obtained by shape class's sizedrag() procedure to circle objects, they would become ellipses, unless the scale values were equal. To prevent this from happening, circle class's resize() procedure first calls shape class's resize() procedure to resize a circle's bounding box. It then finds the center of the resized bounding box, and uses these coordinates as the circle's new center. Finally, it calculates the new radius for the circle, using the smaller of the bounding box's width or height. It then adjusts the circle's bounding box coordinates. Circles are rotated with respect to a point using a standard rotation transform matrix (discussed in Parts 1 and 2 of this series). Circle class's rotate() procedure constructs a rotation matrix and uses it to rotate a circle's center point around the rotation point. This moves the circle, so its bounding box coordinates are updated to reflect its new position.

The last procedure defined for circle class is pointToObject(). This procedure is called to determine the distance between a point (usually the mouse pick point) and an object. Finding the distance between a point and a circle is simple: it is the distance from the point to the circle's center, minus the circle's radius. The distance from the input point to the circle's center is calculated using the distance formula.

Because circle objects are not similar to shape objects, line objects, or polygon objects, we are unable to fully exploit method inheritance to implement circle class. We can only inherit shape class's sizedrag, movedrag, rotatedrag, and extent methods. Rectangles are constrained versions of polygons. Since rectangles are polygons, we can create rectangle class as a subclass of poly class and inherit most of the methods it needs.
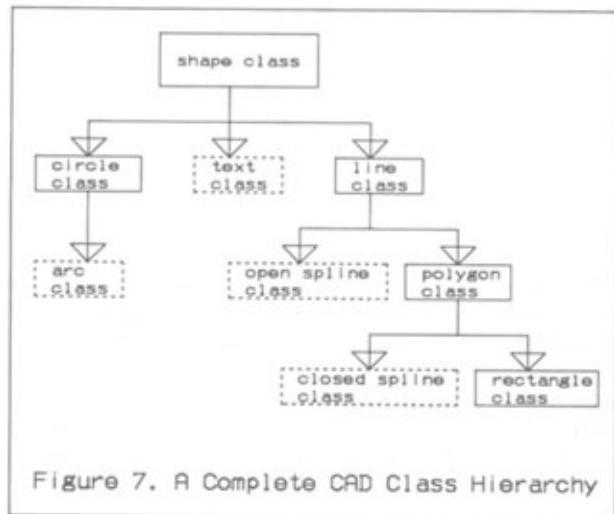
ments. Next, we create the class's private include file and its public include file. We then decide which methods must be implemented for the class, and which can be inherited. Finally, the source code file is created, and we get down to the work of actually coding the class's methods.

Shape class and line class specify all the methods rectangle class needs, so we do not need to add any function pointers to the class structure defined by poly class. Line class specifies the data structure needed to represent lines and polygons, and thus rectangles, so we do not need to add any new data elements to our superclass's object structure. Since rectangle class does not need to add data elements for its objects, we also do not need to define any "atoms" for rectangle objects. In short, rectangle class can



radius = sqrt( dx * dx + dy * dy )
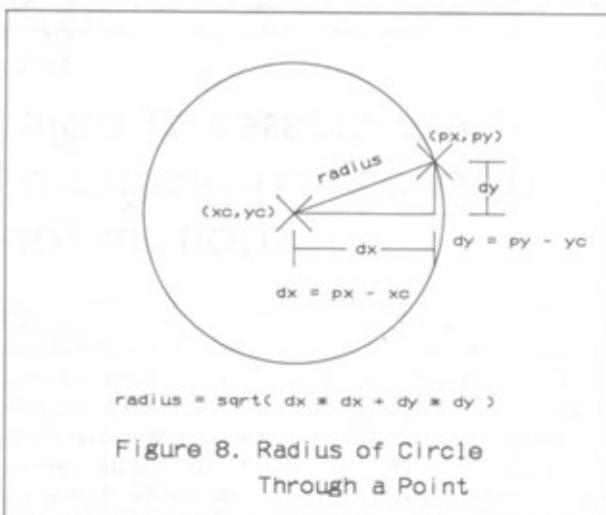
Figure 8. Radius of Circle
Through a Point

simply inherit the class structure, the object structure, and the atoms it needs from its superclasses without extending the structures or adding new atoms. Nevertheless, for the reasons mentioned earlier, we will define empty "class part" and "object part" macros and append them to the class structure and object structure defined by poly class. The macros, and the class structure and object structure for rectangle class and rectangle objects are defined in rectClassP.h, which is in Listing 4. We also export rectangle class's class structure in rectClassP.h so any future subclasses can use it for their superClass pointer. The public include file for rectangle class is shown in Listing 5. It makes a pointer to rectangle class's class structure available to application code. The structure pointer is needed for creating rectangle objects.

At this point, we have defined our data structures and created our class's include files. The next step is to determine which methods to implement. Since we are implementing rectangle class as a subclass of poly class, it will have the methods specified by line class in addition to those specified by shape class. The protocol for shape class requires rectangle class to provide setupClass, setvalue, getvalue, and update methods. Rectangle class can not inherit initialize, deallocate, or copy methods, since these are chained methods. Like circle class, rectangle class does not define any dynamic data elements for its objects (the "points" data element is defined by line class), so it does not need to provide these methods. The remaining methods specified by shape class can be inherited or defined by subclasses. Tables 3 and



Figure 7. A Complete CAD Class Hierarchy

## Implementing Rectangle Class

To implement rectangle class and rectangle objects, we follow the same steps we followed to implement circles. The first step is to decide if rectangle class's class structure needs any function pointers in addition to those specified by its super-classes, and decide if rectangle objects need any new data ele-

4 contain information about the methods specified by line class. From these tables, we see that none of the methods specified by line class are required or chained, and each can be inherited by subclasses. Other than the methods required by shape class, rectangle class can either inherit or redefine the remaining methods specified by shape class, and inherit or redefine all the methods specified by line class.

Polygon objects behave exactly the way we want rectangle objects to behave, with some exceptions. First, the insertdrag method for polygons does not constrain the number of points which can be inserted when new polygons are graphically defined, nor constrain the way edges connecting the points are connected. Rectangles can not have more than four points (actually, we use five points, but the last one is an artificial point), and the lines connecting them must join at right angles. Second, a polygon's

class? Rectangle class's update() procedure does not check for this, so how does it make sure rectangle objects are not resized in such a way that they become distorted? To answer this question, we have to look at what the update() procedures of rectangle class's superclasses do (a production quality protocol description contains this type of information). The source code for line class and poly class is on the disk that accompanies this article, so you can examine their update() procedures in detail. Poly class's update() procedure just calls line class's update() procedure, which detects bounding box value changes. It then calculates x- and y-scale values from the old and new bounding box values, and calls the resize() procedure whose pointer is in the class structure of the object it is updating. For rectangle objects, the class structure pointer is rectClass, and the resize procedure that is called is rectangle class's resize() procedure. Here is the code

---

## New classes of objects are added to our CAD application program by adding an "insert" menu subitem for the new objects.

---

individual points can be dragged and moved. We do not want to allow the points defining rectangles to be moved, since the rectangle will be deformed if its individual points are moved. Third, polygons can be rescaled in such a way that they are distorted. This happens if unequal x- and y-scale values are applied to the polygon's points. If the points defining a rectangle are scaled with unequal x- and y-scale values while it is rotated at an angle other than 0, 90, or 270 degrees, the rectangle will be distorted into a non-rectangular parallelogram. Because of these reasons, we need to define insertdrag and resize methods so we can constrain the points defining rectangles. We also need to disallow dragging and moving of our rectangle objects' individual points by setting pointers to the methods specified by line class to NULL. We can inherit all the remaining methods needed to implement rectangle class. To summarize, the methods we need to define are setupClass, setvalue, getvalue, update, insertdrag, and resize. Having decided what methods to implement, we can now declare rectangle class's procedures, initialize its class structure (_rectClass), and code its procedures. The source code file containing the implementation of rectangle class is in Listing 6. The following paragraphs describe the procedures.

Rectangle class's setup() procedure works exactly like circle class's (and all of our other classes') setup procedures: it makes sure the rectClass structure is initialized, and then resolves inheritance for any of its subclasses. Rectangle class defines only two procedures that can be inherited by subclasses, so only four lines of code are required to support subclass inheritance.

Rectangle class's getvalue(), setvalue(), and update() procedures are extremely simple. They just call their superclass's getvalue(), setvalue(), and update() procedures. The atoms for rectangle objects are defined by line class, so rectangle class does not need to set any values or return any, other than its class name. What if the bounding box for rectangles is changed by shape

from line class's update() procedure which calls rectangle class's resize() procedure ("lineC" is the class structure pointer for the object being updated):

```
    /*
     *  scale with respect to original lower/left
corner    */

    if ( sx != 1.0 || sy != 1.0 )
    {
        if ( lineC->resize )            (void)(*lineC-
>resize)(object,
                                               cpy-
>minx,cpy->miny,sx,sy);
        else               (void)resize(object,cpy-
>minx,cpy->miny,sx,sy);
    }
```

Rectangle class's resize method contains special case code to handle scaling of rectangles that are rotated at angles other than 0, 90, or 270 degrees. Scaling rectangles rotated at angles other than these with unequal x- and y-scale values causes them to become non-rectangular. You can observe this in our mini-CAD program by creating a rectangular polygon, rotating it, and then scaling it. To prevent this from happening, the resize() procedure finds the angle of rotation of a rectangle by finding the angle of rotation of its original bottom edge. If the angle is 0, 90, or 270 degrees, the rectangle is resized using whatever scale values were specified. Otherwise, both the x- and y-scale values are set equal to the smaller of the two, and the rectangle is then resized using the equal scale values. The resized rectangle is then moved so its center is located where it would have been if it had been resized using the unconstrained scale values. Although rectangle class

implements a resize() procedure, it capitalizes on the resize() procedure of its superclass by calling it to do the actual scaling transformation. Rectangle class's resize() procedure just imposes constraints on how rectangles are rescaled.

The list of points defining a rectangle object is created in rectangle class's insertdrag() procedure. This procedure obtains the initial point coordinates by creating a drag rectangle from the pick point, and using dragDrawRect() to process the drag interaction with handleDrag(). When handleDrag() returns, the corner coordinates of the drag box are used to generate the list of points defining the rectangle.

That's basically all the code we have to write to implement rectangles, thanks to inheritance! We have one more task to perform before we can use our new objects: we need to add them to our mini-CAD program. This is the easy part.

## Adding New Classes to the CAD Program

New classes of objects are added to our CAD application program by adding an "insert" menu subitem for the new objects. This is done in the file menu.c. We need to include the new classes' public include files in menu.c, and define three structures for each new class of objects. The three structures are an IntuiText structure containing the menu item name for the object, a miData_t structure containing the data sent to the menu item's pick event procedure, and a myMenuItem_t structure containing the menu item's definition. The three structure definitions for rectangles are:

```
static struct IntuiText insTxt[]  =
  {
       { 2,1,JAM2,2,1,NULL,"Rectangle",NULL },
:


                   :
                   :
  };


   static miData_t insRectData =   { dragAndInsert,
0, (void*)&rectClass };


   static myMenuItem_t insertItm[] =
  {
       { /* "rectangle" sub-item        */
         { &(insertItm[1].mi),0,0,0,0,
           ITEMTEXT|ITEMENABLED|HIGHCOMP,0,


       (APTR)&insTxt[0],NULL,NULL,NULL,MENUNULL
},
         { miSetAction,(void*)&insRectData  }
       },


                   :
                   :

  };
```

After these definitions are added to menu.c, the program is recompiled and relinked, and voila!, our new objects can be put to work. Consider the significance of what we have just done. We added new objects to our CAD application without modifying any of its code! Being able to extend an existing system without modifying its source code is a major benefit of object-oriented systems.

## Conclusion

This has been a long series of articles. We have explored several different topics, discussed techniques for implementing object-oriented CAD systems, and developed some useful software along the way. In Part 1, we developed a library of 2-D CAD transform procedures. In Part 2, we discussed techniques and developed procedures for attaching event handlers to Intuition input objects. In Part 3, we  saw how to implement an object-oriented CAD system in "vanilla C", and in this final article, we saw how to use our object framework to create brand new objects for our CAD program. Although our mini-CAD program is by no means complete, it implements the core functionality found in all CAD systems. With a little work, you should be able to use the code which accompanies this article and the techniques we have discussed to turn our mini-CAD program into a complete object-oriented CAD system! Have fun!

☑

**The listings and all necessary files for CAD Application Design Part 4 can be found on the AC's TECH disk.**

**Please write to
Forest W. Arnold
c/o AC's TECH
P.O. Box 2140
Fall River, MA  02722-2140**

# Developer's Tools

# Writing an Effective Press Release

## by Jeff James

After months of refining code, squashing bugs, rewriting documentation, and creating attractive packaging for your program, you're finally ready to release your product to the Amiga-using public. Now you can dash off a press release and start sending review copies out to your favorite Amiga magazines, right?

Admittedly, creating a press release is often the last thing on a programmer's mind. Some developers believe that a polished, well-designed piece of software should be enough to guarantee some press coverage. While good software does happen to sell itself, a thoughtfully written, timely delivered press release serves an important purpose. The press release is an important tool, a tool which can help you announce new products and get those products more media coverage. A press release can be used in other instances as well, such as when you ship a major software update, drastically change prices, or when your company undergoes a major change in management or ownership.

## Why Should I Use a Press Release?

A good way to understand the importance of a press release is to look at how the media uses them. In the case of the Amiga press, a press release alerts them to new items, which are assigned to a New Products Editor. This editor is often the first person to look at your product. Using your press release as reliable background information, this editor writes a short description of your product for the New Products section. From here, your program—with press release attached—travels to another editor, usually a Review Editor or Associate Editor. This person looks at your product and reads your press release, using both to decide whether to assign the product to a staff editor or freelance writer for review, or to hold the product for later examination.

At each of these stages, the value of a press release cannot be overstated. The New Products editor draws information from your release to write a short description of your product; the Associate editor uses the press release to help gauge how important your product is and how it should be reviewed; and finally, the actual reviewer of the product often uses a press release to become familiar with the main features of your product. If your press release is filled with spelling and grammatical errors, or lists incorrect information, it makes the job of each one of these individuals more time-consuming. Remember that journalists are always on a deadline: time spent on the phone clarifying a confusing or ambiguous press release translates into less time for the reviewer to spend evaluating your product. If the editor can't reach the person listed as a contact on your press release (or if no contact is listed), your product might be put aside until questions can be answered and ambiguities clarified. Your press release is often the only communication an editor might see from your company, so it's important to make a good impression.

## Is There a Right Way to Write a Press Release?

Although there isn't a set of rules cast in stone someplace on how to write a good press release, here are some generally accepted guidelines dealing with the basic structure and layout of a press release:

1. Use good quality, letter-size (8.5" x 11") paper. Double-space all text, using just one column on each page. Set top, bottom, left, and right margins at 1".

2. Put your company logo, address, and telephone number in the upper left corner on the first page of the release. Try to limit press releases to two pages, and print on only one side of each page.

3. In the upper right corner of the first page, parallel to your company information, list a contact name and phone number. This should be a person an editor can contact for clarification or questions on information included in your press release. Make sure the person listed is aware of the press release and can accurately respond to any inquiries.

4. Use a headline to quickly convey the main content of your press release. Center the headline and place it directly over the body text of your press release on the first page.

5. Place the date at which the information in the press release can be used in the upper left corner of the first page, between your company information and the headline. You'll normally want to put "For Immediate Release" in this spot, unless you want the information released at a certain date. In that case, use this format: "For September 26 Release." Keep in mind that most magazines are published monthly, so alter release dates accordingly.

6. On the first line of your main text, print the city and state from which the release is originating in capital letters, followed by a release date. Use this format: "FALL RIVER, MASSACHUSETTS: August 3rd, 1992 —"

7. Avoid using excessive computer jargon. Although most editors are exceptionally familiar with computer terms, take the time to adequately explain uncommon acronyms and other rarely used language.

8. If your release announces a new product, be sure to include an extensive list of program requirements. This information should usually be saved for the last paragraph of your press release.

9. If more than one page, put the word "MORE," centered, on the bottom of the first page. Signify the end of the press release by placing the symbols "###" or the number "30" at the bottom of the last page. Output the press release to a high-quality printer, preferably an ink-jet or laser. (The sample press release on p. 79 uses all the suggestions listed above.)

Again, these are only rough guidelines — as long as your release is short (two pages or less), tersely written, and supplies all the necessary information, you can deviate somewhat from this format. Once you've chosen the physical layout of your press release, it's time to look at what goes into a press release. What follows are some general suggestions on writing a press release to announce a new product.

## Guidelines for Grammarians

If you're shipping a new product, avoid the temptation to make your press release a giant feature list. Think about what makes your product differ from competitive products, and concentrate on those. For example, if your new product is a word processor, don't spend valuable space discussing its word-wrap feature and ability to open and save documents—focus on what makes your product unique, perhaps its ability to read a document aloud or extensive ARexx support.

After you've decided what you're going to include, you can begin writing your release. It's important at this point to mention the twin bugbears facing every budding writer of press releases—grammar and spelling. Even good, experienced writers occasionally commit some errors, so keep a dictionary handy and use a spell checker often. Many of the books listed at the end of this article can be very helpful with grammar, particularly *The Elements of Style*, by Strunk and White, and *The Chicago Manual of Style*, by the University of Chicago press. *The Associated Press Stylebook and Libel Manual* is another good text, used by journalists themselves to find rules on spelling, punctuation, and grammar.

When writing your press release, it's important to write in the active voice as opposed to the passive voice. For example, "AcmeSoft's C++ Professional supports AmigaDOS 2.0" is in the active voice; "AmigaDOS 2.0 is supported by AcmeSoft's C++ Professional" is in the passive voice. Write by "doing" (active) instead of "showing" (passive). Using the active voice in your press release shortens sentences and makes for easier reading. You'll also want to write your press release in what journalists call the "inverted pyramid" writing style. Visualize a real pyramid flipped upside down—the wide, thick base of the pyramid is at the top. As you move lower, the pyramid gets smaller and smaller, eventually tapering off to a tiny point. Just like this topsy-turvy pyramid, you'll want to put the "biggest," most important ideas at the top, or beginning of your press release, followed by the next most important, and so on, until your release draws to a close. Used by print journalists worldwide, this form of writing manages to pack the most important information into as small a space as possible. The first sentence of your press release, called a "lead," should contain the most important information you are trying to convey. For example, if your company just released a new game, your lead would look something like this: "Gamesoft has released Midway Monsters, a new football game featuring rotoscoped animation of real players, for the Amiga computer." In just one sentence, you've introduced your company, announced a new product, stated a unique feature of that product, and mentioned which computer that product operates on. If you aren't very familiar with this style of writing, just drop by your local library or bookstore and ask for books on newswriting or print journalism.

## Other Points to Remember

When releasing new product information to a publication, it's a good idea to send two review copies of your product—sturdily enclosed within adequate packing material—along with your press release. Why two copies? One copy stays with the magazine's editorial staff, while the other is often mailed out to a freelance author for review. For press releases which don't involve new products, a large 9" x 11" envelope with the press release (and possibly a business card) will suffice.

As bad as sending a poorly crafted press release to the media can be, sending too many press releases can do even more damage. Just as in the Aesop's Fable where the boy cries "wolf" one too many times, flooding an editor's desk with press releases of marginal newsworthiness can result in a truly important announcement of yours being overlooked. A press release dealing with such things as minor personnel changes, new packaging for your product, or the forthcoming marriage of your lead programmer may be very important to you, but of little interest to the Amiga community at large. Limit your press releases to new product announcements, significant product updates, major organizational changes, and other information of equal importance.

Finally, it's always wise to send your press kit addressed to a specific individual as opposed to a generic job title. Addressing your communication specifically to one individual improves the chances of your product getting noticed. If necessary, call the magazine beforehand and find out the full name of the person in question (be sure to get the correct spelling, too). It does work. Just think how you feel when you receive a letter addressed to "current resident," and you'll see why this step is effective.

These suggestions obviously cannot replace a structured course on public relations writing. If you're new to writing press releases, several of the books listed at the end of this article are excellent reading material. Taking a course on public relations or persuasive writing at a local community college can also be helpful. Obviously, a press release can't work miracles. As a software developer, your first priority should be to create a solid, reliable program that people would want to purchase. However, in an increasingly competitive Amiga market, a well-written press release can help tip the scales in your favor.

### Suggested Reading:

The Chicago Manual of Style, 13th ed. University of Chicago Press, Chicago, 1982. ISBN 0-226-10390-0, $37.50

The Elements of Style, 3rd ed. William Strunk Jr. and E.B. White. Macmillan Publishing, New York, 1979. ISBN 0-02-418200-1 $5.95

Business Writers Handbook, 3rd ed. Brusaw, Alred, Oliver. St. Martin's Press, New York, 1987. ISBN 0-312-10958-X $19.95

News Reporting and Writing, 3rd ed. Brooks, Kennedy, Moen, Ranly. St. Martin's Press, New York, 1988. ISBN 0-312-00279-3 $23.95

How to Write a Dynamic Press Release. The Communication Workshop, 217 East 85th Street, New York, NY 10028, (516) 767-9590, $8.00

The Associated Press Stylebook and Libel Manual. Norm Goldstein,Ed. Addison-Wesley Publishing, New York, 1992. ISBN 0-201-56760-1, $11.95

Contact: Jim Acme
508-678-4800, ext. 123

# A.S.I.

AcmeSoft, Inc
1234 Serendipity Drive, Suite 300
Fall River, MA 02720-7160
508-678-4800

FOR IMMEDIATE RELEASE

## AcmeSoft Releases C++ Professional Compiler for the Amiga

FALL RIVER, MASSACHUSETTS: August 3rd, 1992 — AcmeSoft Inc. (ASI) has just released the C++ Professional Compiler (C++ Pro) for the Amiga computer. C++ Pro is a powerful program development system which brings the latest advances in object-oriented programming technology to Amiga software developers.

"ASI's C++ Pro for the Amiga has cut our program development time by 30%," says Gary Buss, lead programmer for Penguin Software. "C++ Pro gives our programmers a powerful development system for creating the best in Amiga game software."

ASI's C++ Pro offers fast compile speed and full support for the new AmigaDOS 2.0 "look and feel." C++ Pro also includes:

• Acme Debugger • Acme Compiler • Full support for ANSI C

ASI's C++ Pro has a suggested retail price of $149.99 ($169.99 CDN) and operates on all Amiga models with 1 MB of RAM and Kickstart 2.0 or higher. A hard drive and color monitor are recommended. ASI is a diversified software company that develops and publishes Amiga productivity software worldwide.

-30 -

# AC Order Form!

Name_____

Address_____

City_____ State _____ ZIP_____

Charge my ☐ Visa ☐ MC  #_____

Expiration Date _____ Signature _____

**MasterCard**   **VISA**

Please circle to indicate this is a **New Subscription** or a **Renewal**

| | | |
|---|---|---|
| **One Year of Amazing!** | **Save over 43%** <br> 12 monthly issues of the number one resource to the Commodore Amiga, *Amazing Computing* at a savings of over $20.00 off the newsstand price! | ☐ US $27.00 <br> ☐ Canada/Mexico $34.00 <br> ☐ Foreign Surface $44.00 |
| **One Year of AC SuperSub!** | **Save over 45%** <br> 12 monthly issues of Amazing Computing PLUS *AC GUIDE AMIGA* 2 Product Guides a year! A savings of over $30.00 off the newsstand price! | ☐ US $37.00 <br> ☐ Canada/Mexico $54.00 <br> ☐ Foreign Surface $64.00 |
| **Two Years of Amazing!** | **Save over 56%** <br> 24 monthly issues of the number one resource to the Commodore Amiga, Amazing Computing at a savings of over $53.80 off the newsstand price! | ☐ US $41.00 <br> (sorry no foreign orders available at this frequency) |
| **Two Years of AC SuperSub!** | **Save over 56%** <br> 24 monthly issues of Amazing Computing PLUS *AC GUIDE AMIGA* 4 Product Guides! A savings of over $75.60 off the newsstand price! | ☐ US $59.00 <br> (sorry no foreign orders available at this frequency) |
| **One Year of AC's TECH!** | **PLUS!** AC TECH AMIGA <br> 4 quarterly issues of the first Amiga technical reference magazine with disk! | ☐ US $43.95 <br> ☐ Canada/Mexico $47.95 <br> ☐ Foreign Surface $51.95 |

Please circle any additional choices below:          *(Domestic and Foreign air mail rates available on request)*

**Amazing Computing Back Issues**: $5.00 each US, $6.00 each Canada and Mexico, $7.00 each Foreign Surface. Please list issue(s)_____

**Amazing Computing Back Issue Volumes:**
Volume 1-$19.95*  Volume 2, 3, 4, 5, or 6-$29.95* each
*All volume orders must include postage and handling charges: $4.00 each set US, $7.50 each set Canada and Mexico, and $10.00 each set for foreign surface orders. Air mail rates available.

**AC TECH AMIGA**   **Single issues just $14.95!** V1.1 (Premiere), V1.2, V1.3, V1.4, V2.1, or V2.2
**Volume One complete— $45.00!** (All Four issues)

**Freely Distributable Software – Subscriber Special (yes, even the new ones!)**

| 1 to 9 disks | $6.00 each |
|---|---|
| 10 to 49 disks | $5.00 each |
| 50 to 99 disks | $4.00 each |
| 100 or more disks | $3.00 each |

$7.00 each for non subscribers     (three disk minimum on all foreign orders)

**Amazing on Disk:**

| | |
|---|---|
| AC#1 ...Source & Listings V3.8 & V3.9 | AC#2 ...Source & Listings V4.3 & V4.4 |
| AC#3 ...Source & Listings V4.5 & V4.6 | AC#4 ...Source & Listings V4.7 & V4.8 |
| AC#5 ...Source & Listings V4.9 | AC#6 ...Source & Listings V4.10 & V4.11 |
| AC#7 ...Source & Listings V4.12 & V5.1 | AC#8 ...Source & Listings V5.2 & 5.3 |
| AC#9 ...Source & Listings V5.4 & V5.5 | AC#10 ...Source & Listings V5.6 & 5.7 |
| AC#11 ...Source & Listings V5.8, 5.9 & 5.10 | AC#12 ...Source & Listings V5.11, 5.12 & 6.1 |
| AC#13 ...Source & Listings V6.2 & 6.3 | AC#14 ...Source & Listings V6.4, & 6.5 |
| AC#15 ...Source & Listings V6.6, 6.7, 6.8, & 6.9 | AC#16 ...Source & Listings V6.10, 6.11, 6.12, 7.1, 7.2, & 7.3 |

Please list your Freely Redistributable Software selections below:

**AC Disks** _____
*(numbers 1 through 16)*

**AMICUS** _____
*(numbers 1 through 26)*

**Fred Fish Disks** _____
*(numbers 1 through 660)*

Subscription: $_____

Back Issues: $_____

*AC's TECH*: $_____

PDS Disks: $_____

Total:     $_____
(subject to *applicable* sales tax)

### Complete Today, or telephone 1-800-345-3360 now!

*You may FAX your order to 1-508-675-6002*

Please complete this form and mail with check, money order or credit card information to:

Please allow 4 to 6 weeks for delivery of subscriptions in US.

**P.i.M. Publications, Inc.**
**P.O. Box 2140**
**Fall River, MA 02722-0869**

# Quarterback 5.0

## The Next Generation In Backup Software

Quarterback 5.0 - © 1992 Central Coast Software

□ Quarterback

```
/* Backup started Feb 11, 1992 at 10:55:02 AM */
System2.0
  C
     AddBuffers
     Arc
     Assign
     Avail
     BindDrivers
     Break
     ChangeTaskPri
     ConClip
     Copy
     CPU
     Date
     Delete
     Dir
     DiskChange
     DiskDoctor
     DiskSalv
     Ed
     Edit
     Eval
     Execute
     Filenote
     IconX
     Info
     Install
     IPrefs
```

**Backup in progress.**

[ Pause ]   [ Abort ]

DF0: Writing #1
DF1: Not available
DF2: Ready
DF3: Ready

Completed:        4%
Files:           25
Bytes:      178,560

Tagged:
Files:          559
Bytes:    4,599,613